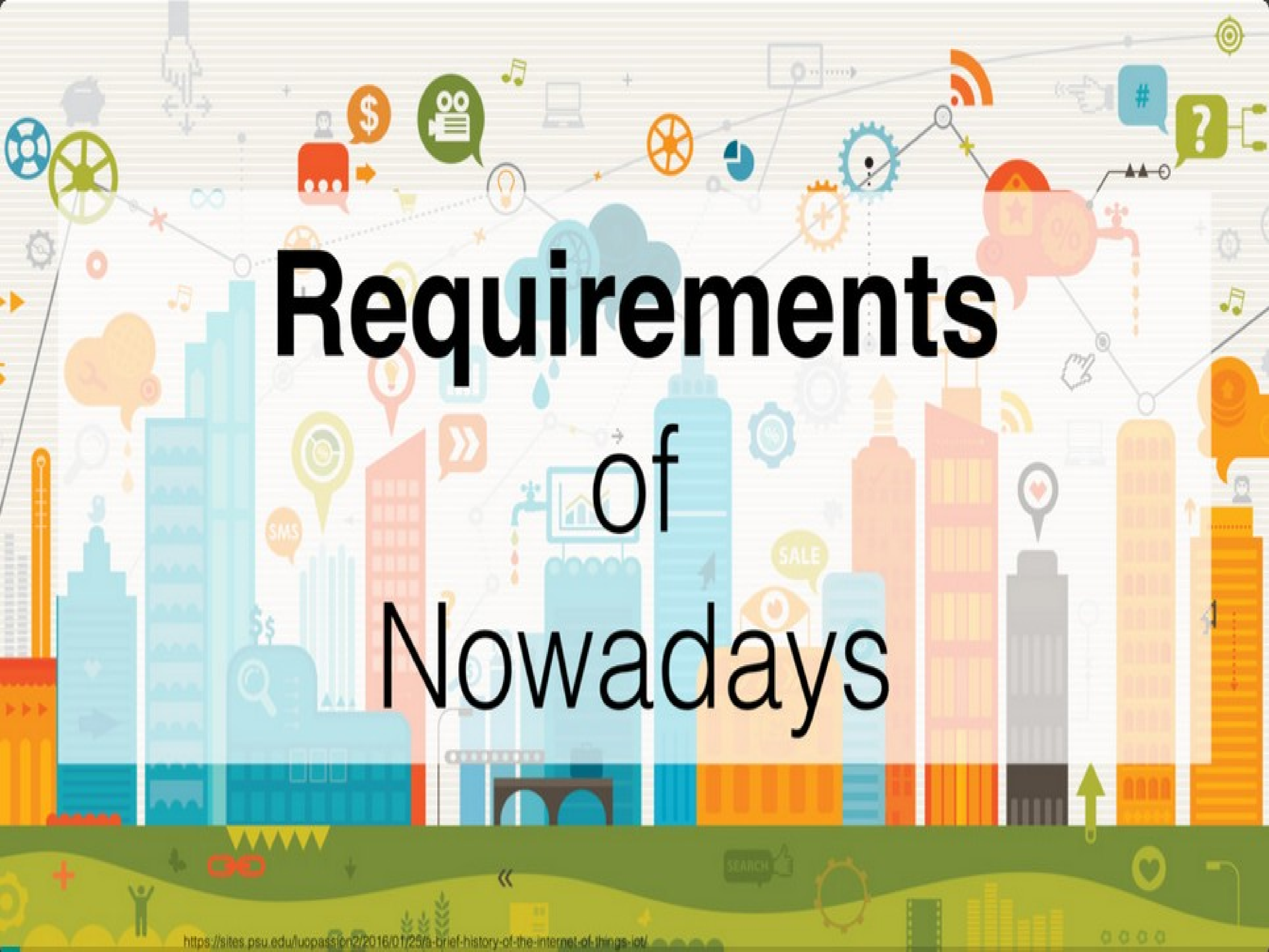
Reactive Programming Introduction
By Mohit Kumar

Agenda

- Why Reactive?
- ReactiveStreams
- ReactiveProgramming-Introduction

So, **why** Reactive?[1]



Requirements of Nowadays

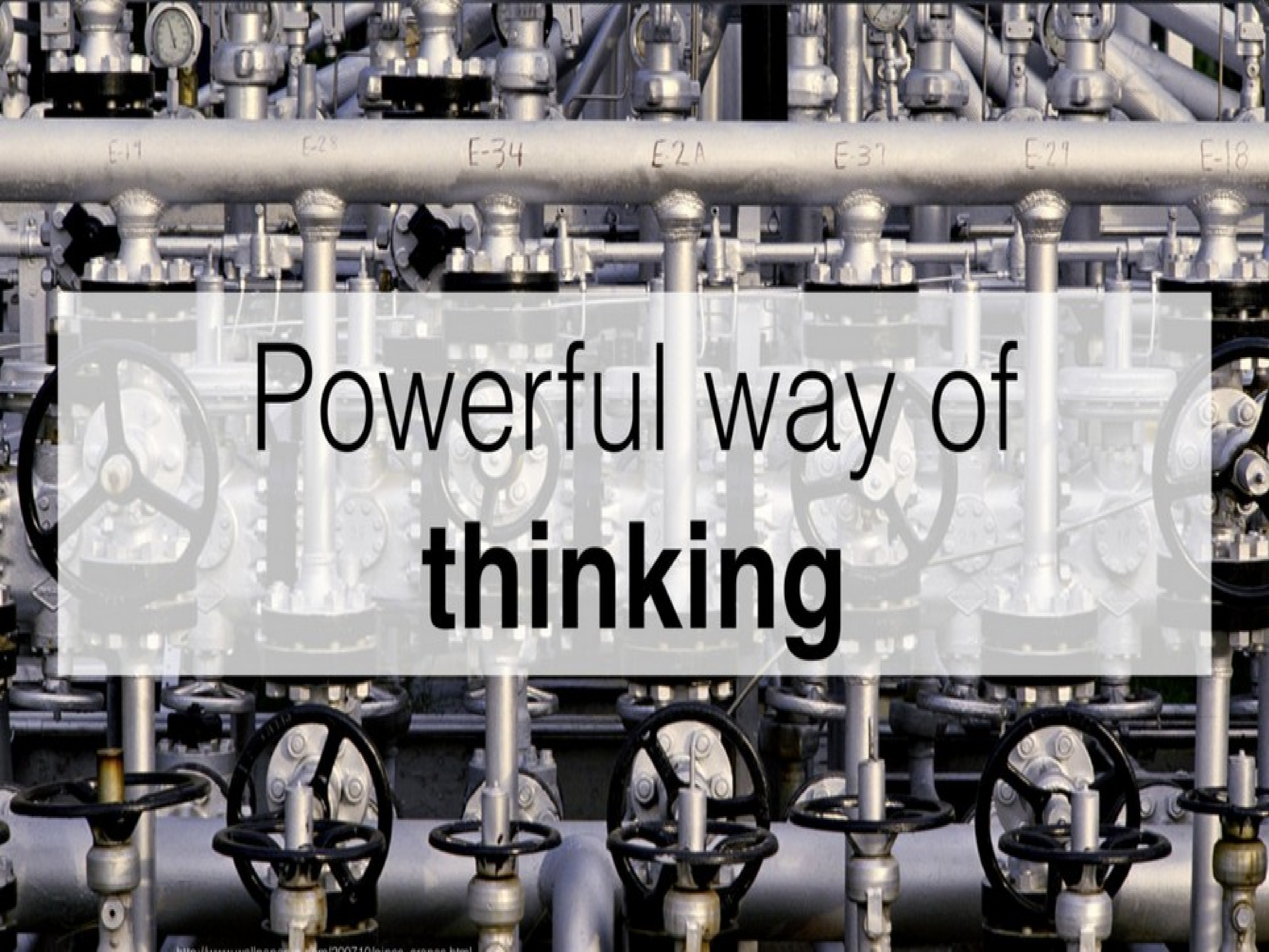
https://sites.psu.edu/luopassion2/2016/01/25/a-brief-history-of-the-internet-of-things-iot/

Lean resource management

Asynchrony

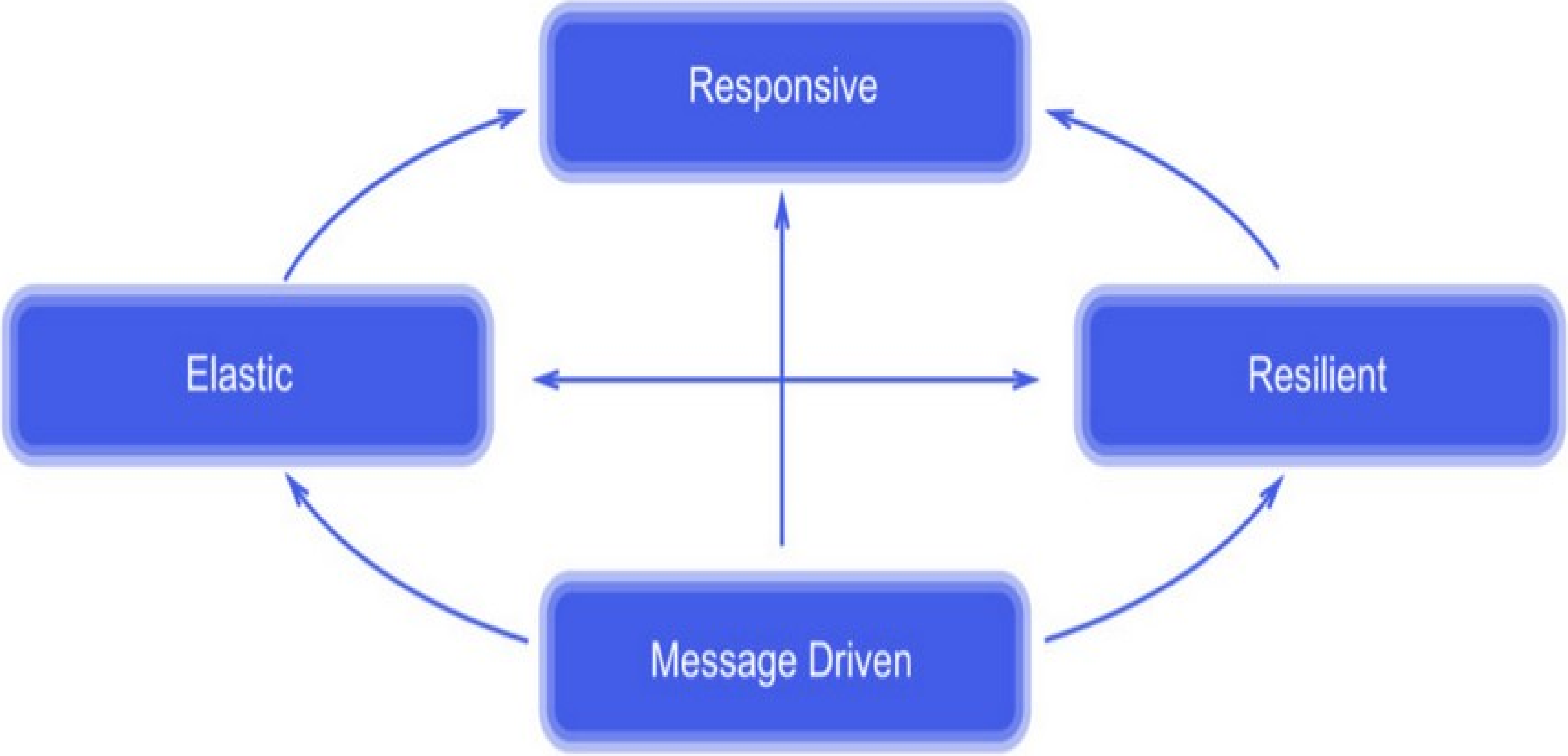
Fault Tolerance

Powerful way of **thinking**

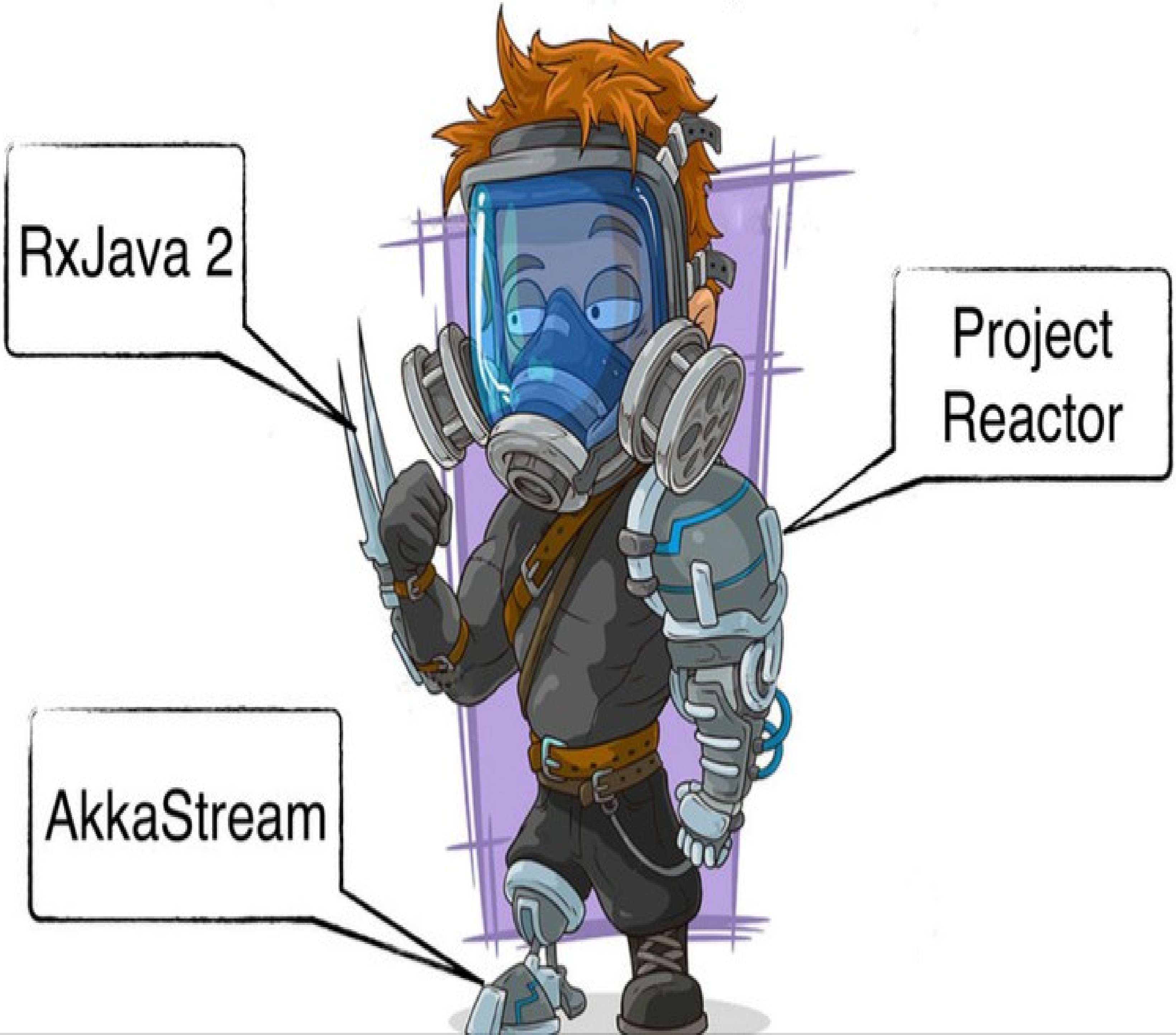


http://www.wall[illegible]m/200710/[illegible].html

Reactive-Manifesto[2]



Reactive Streams Specified



RxJava 2

Project Reactor

AkkaStream

Reactive Streams[4]

What is the Purpose?

Backpressure

Common API

Publisher

Subscriber

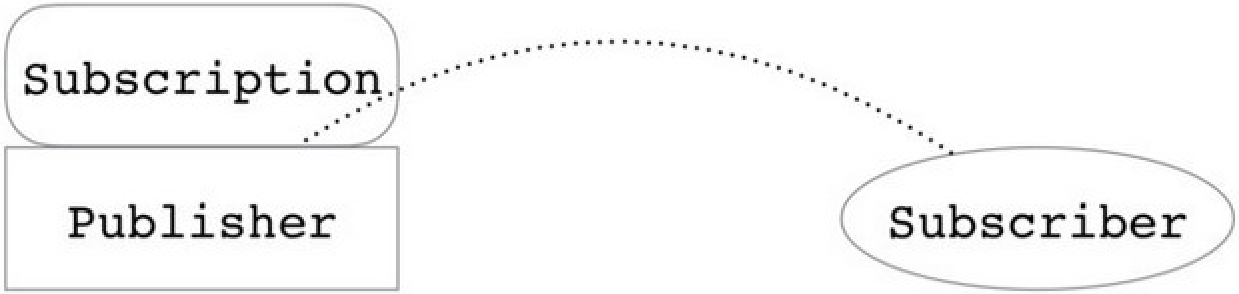
```
public interface Publisher<T> {
  public void subscribe(
      Subscriber<? super T> s
  );
}
```

Publisher

Subscriber

```
public interface Subscriber<T> {
  public void onSubscribe(Subscription s);
  public void onNext(T t);
  public void onError(Throwable t);
  public void onComplete();
}
```

Publisher
Subscriber

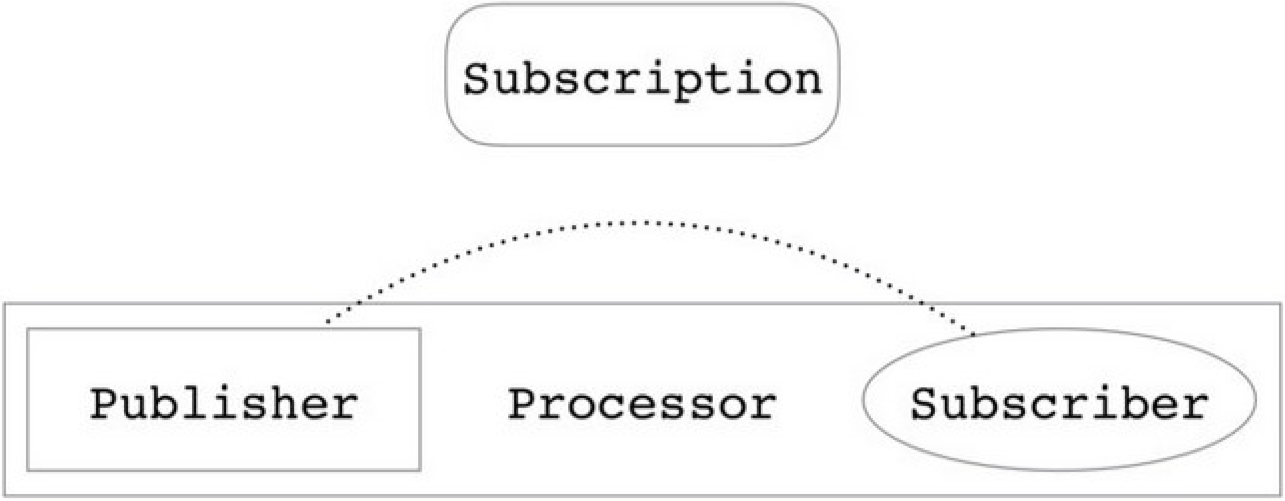

Subscription
Publisher
Subscriber

Subscription

```
public interface Subscription {
    public void request(long n);
    public void cancel();
}
```

Subscription

```
public interface Subscription {
    public void request(long n);
    public void cancel();
}
```


Subscription
Publisher
Processor
Subscriber

```
public interface Processor<T, R>
  extends Subscriber<T>, Publisher<R>
{}
```

Processor

ReactiveProgramming-Introduction

Reactive Programming

- The mental model for a Future and CompletableFuture is that of a computation that executes independently and concurrently. The result of the Future is available with get() after the computation completes.
  - Thus, Futures are one-shot, executing code that runs to completion only once.
- By contrast, the mental model for reactive programming is a Future-like object that, over time, yields multiple results.
  - Consider two examples,
    - starting with a thermometer object. You expect this object to yield a result repeatedly, giving you a temperature value every few seconds.
    - the listener component of a web server waits until an HTTP request appears over the network and provides data from the request. Then other code can process the result: a temperature or data from an HTTP request.
      - Then the thermometer and listener objects go back to sensing temperatures or listening before potentially yielding further results.

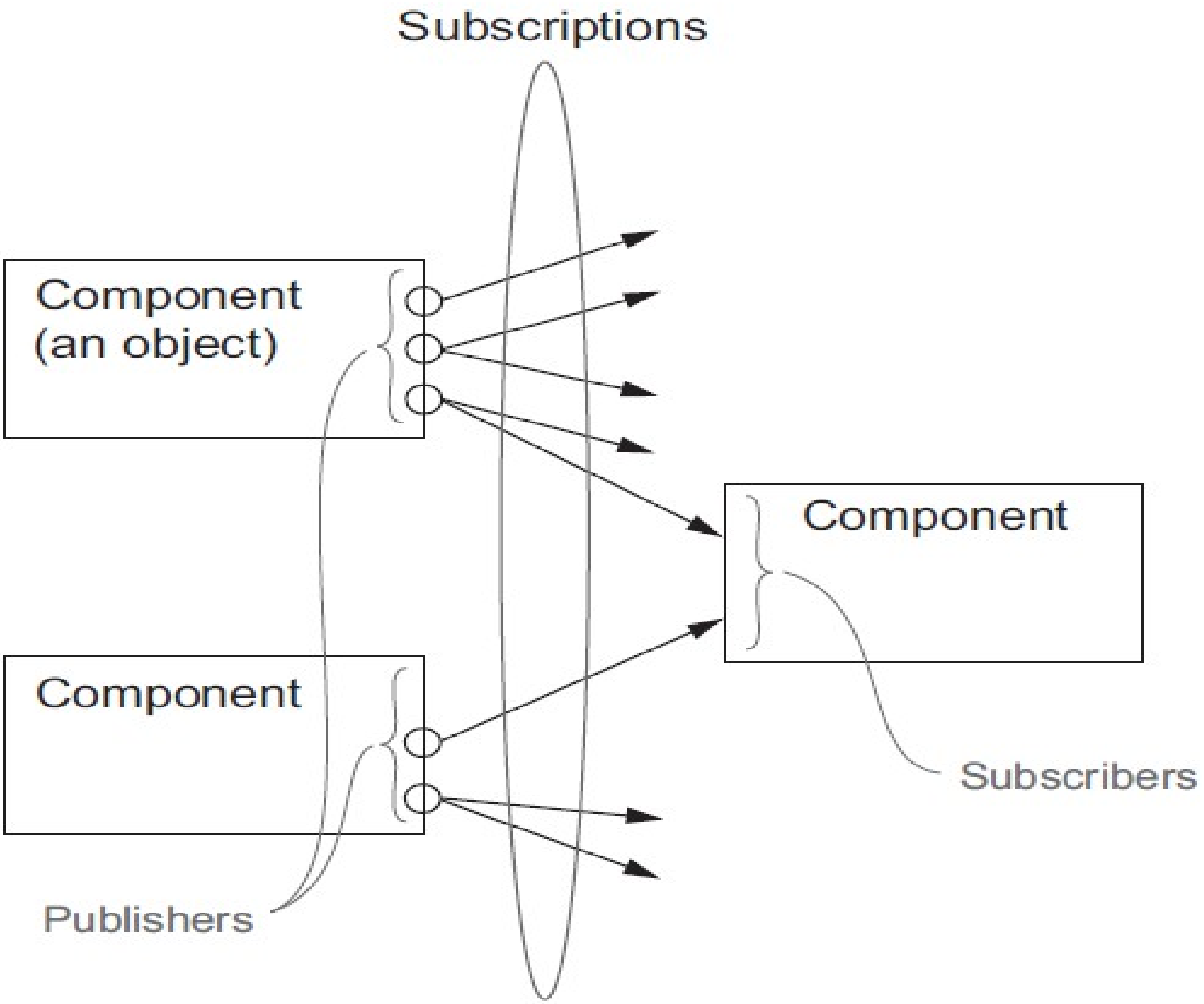
Reactive Programming

- Differentiating factor
  - The core point is that these examples are like Futures but differ in that they can complete (or yield) multiple times instead of being one-shot.
  - Another point is that in the second example, earlier results may be as important as ones seen later, whereas for a thermometer, most users are interested only in the most-recent temperature.

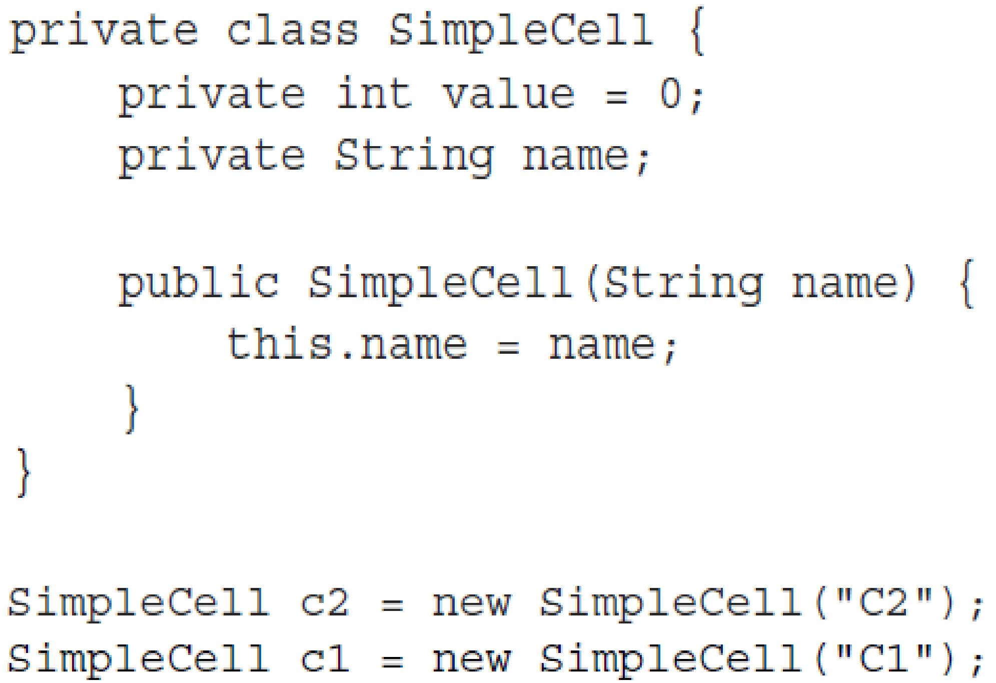
Reactive Programming

- Java 9 models reactive programming with interfaces available inside java.util.concurrent.Flow and encodes what’s known as the publish-subscribe model (or protocol, often shortened to pub-sub). There are three main concepts:
  - A publisher to which a subscriber can subscribe.
  - The connection is known as a subscription.
  - Messages (also known an events) are transmitted via the connection.

Reactive Programming

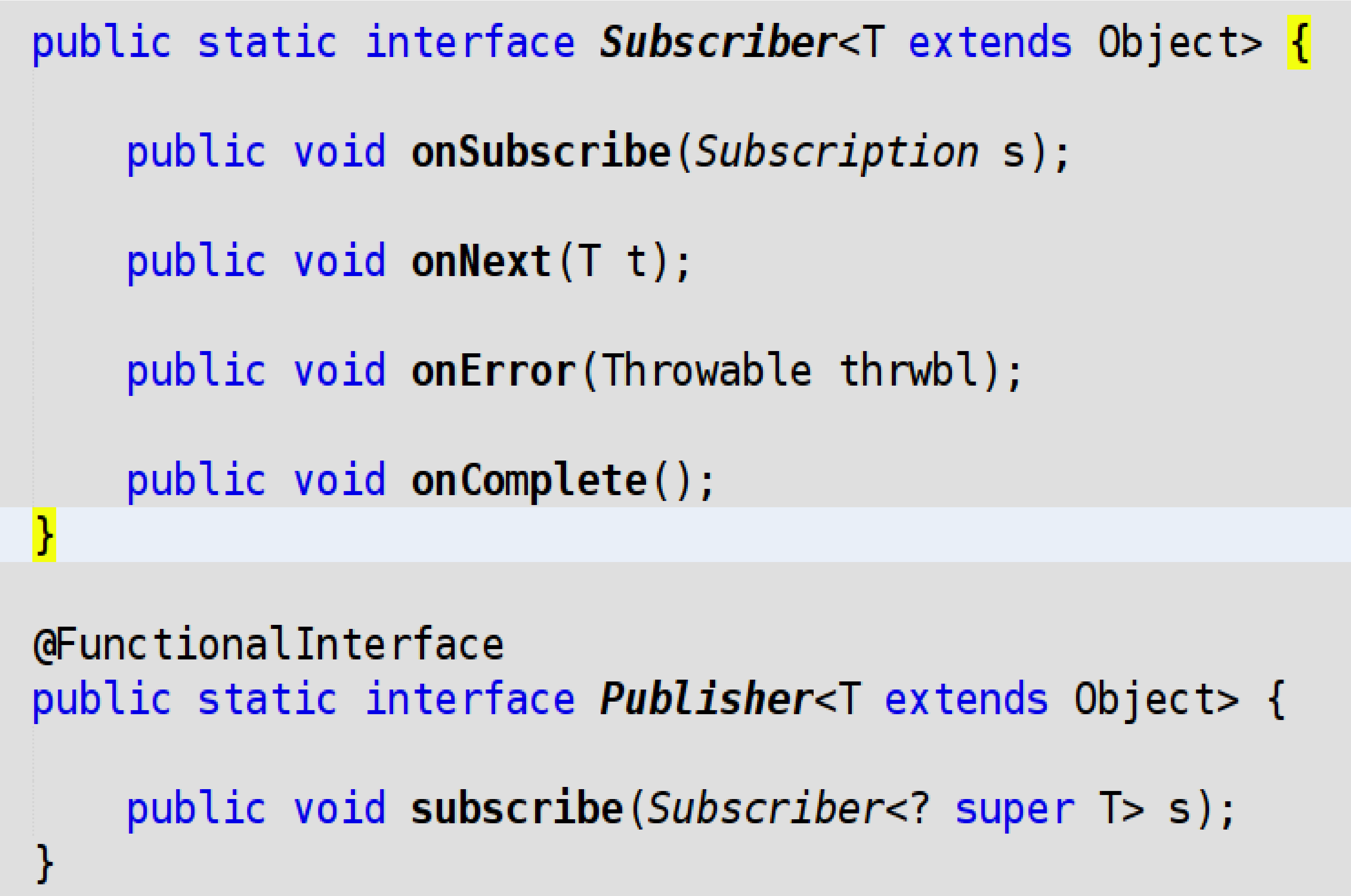


Reactive Programming:Simple Example



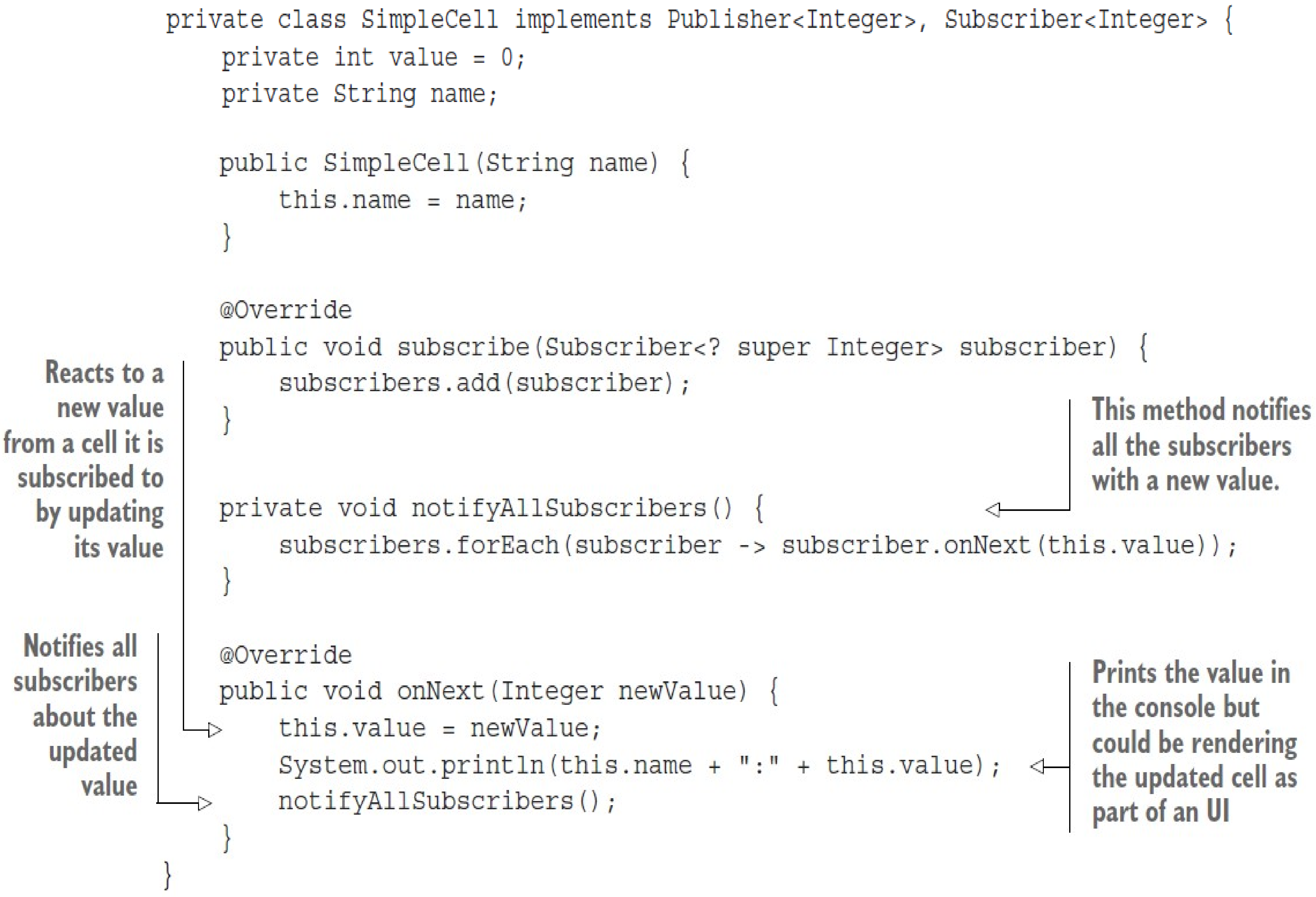
```
private class SimpleCell {
    private int value = 0;
    private String name;

    public SimpleCell(String name) {
        this.name = name;
    }
}

SimpleCell c2 = new SimpleCell("C2");
SimpleCell c1 = new SimpleCell("C1");
```

Reactive Programming:Flow



```
public static interface Subscriber<T extends Object> {

    public void onSubscribe(Subscription s);

    public void onNext(T t);

    public void onError(Throwable thrwbl);

    public void onComplete();
}

@FunctionalInterface
public static interface Publisher<T extends Object> {

    public void subscribe(Subscriber<? super T> s);
}
```

Reactive Programming:Simple Example



```
private class SimpleCell implements Publisher<Integer>, Subscriber<Integer> {
    private int value = 0;
    private String name;

    public SimpleCell(String name) {
        this.name = name;
    }

    @Override
    public void subscribe(Subscriber<? super Integer> subscriber) {
        subscribers.add(subscriber);
    }

    private void notifyAllSubscribers() {
        subscribers.forEach(subscriber -> subscriber.onNext(this.value));
    }

    @Override
    public void onNext(Integer newValue) {
        this.value = newValue;
        System.out.println(this.name + ":" + this.value);
        notifyAllSubscribers();
    }
}
```

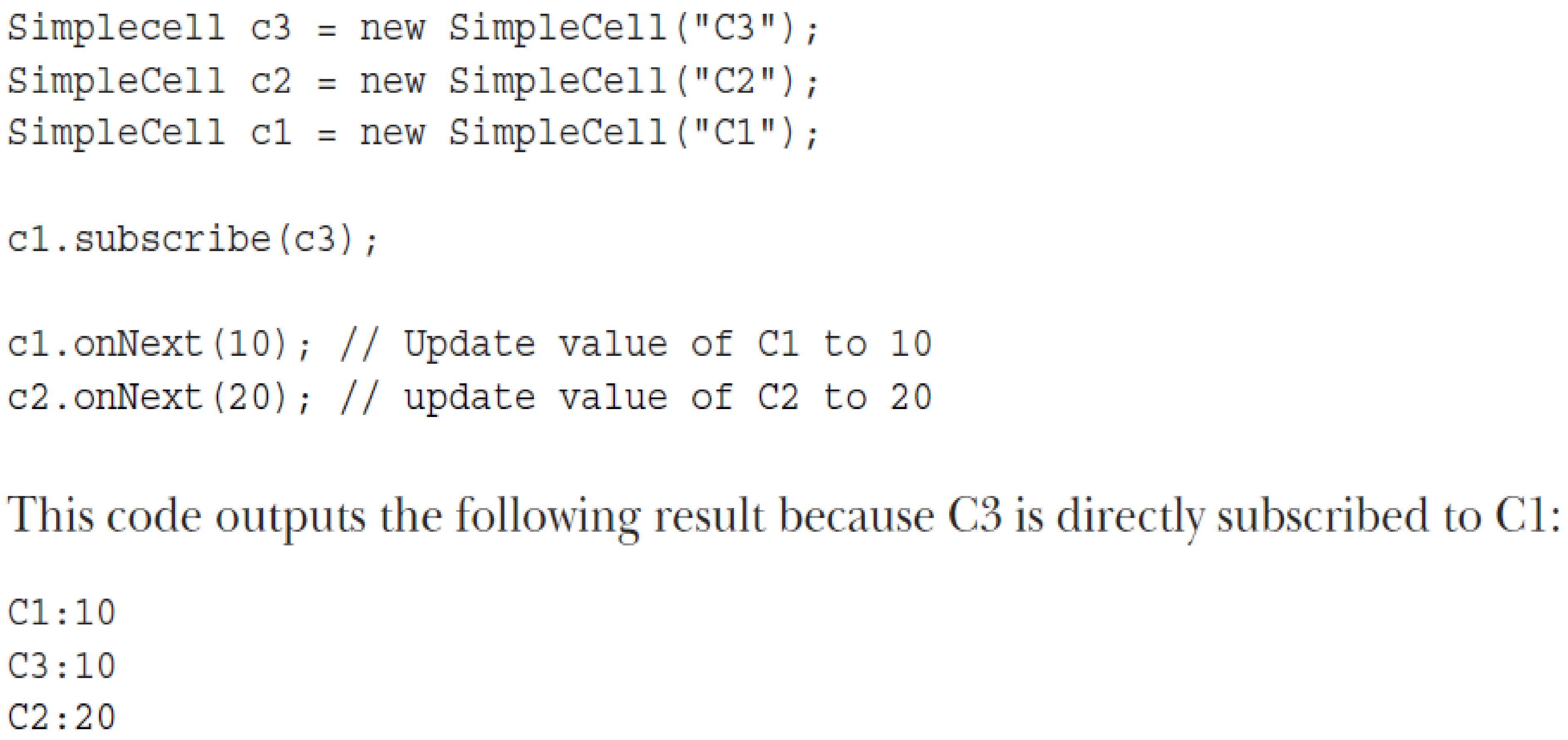
This method notifies all the subscribers with a new value.

Reacts to a new value from a cell it is subscribed to by updating its value

Notifies all subscribers about the updated value

Prints the value in the console but could be rendering the updated cell as part of an UI

Reactive Programming:Simple Example

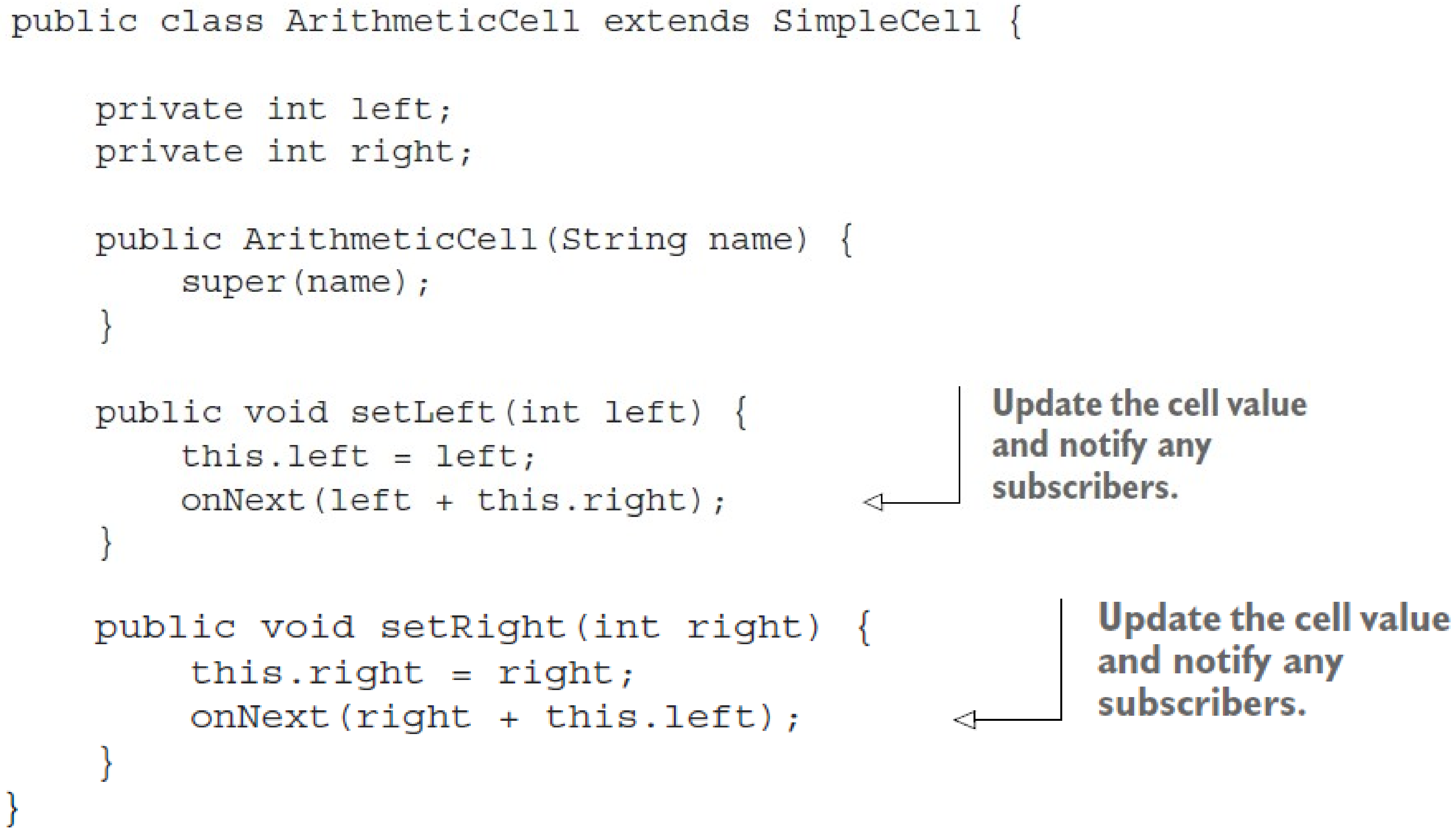


```
Simplecell c3 = new SimpleCell("C3");
SimpleCell c2 = new SimpleCell("C2");
SimpleCell c1 = new SimpleCell("C1");

c1.subscribe(c3);

c1.onNext(10); // Update value of C1 to 10
c2.onNext(20); // update value of C2 to 20
```

This code outputs the following result because C3 is directly subscribed to C1:

```
C1:10
C3:10
C2:20
```

Reactive Programming:Simple Example-2

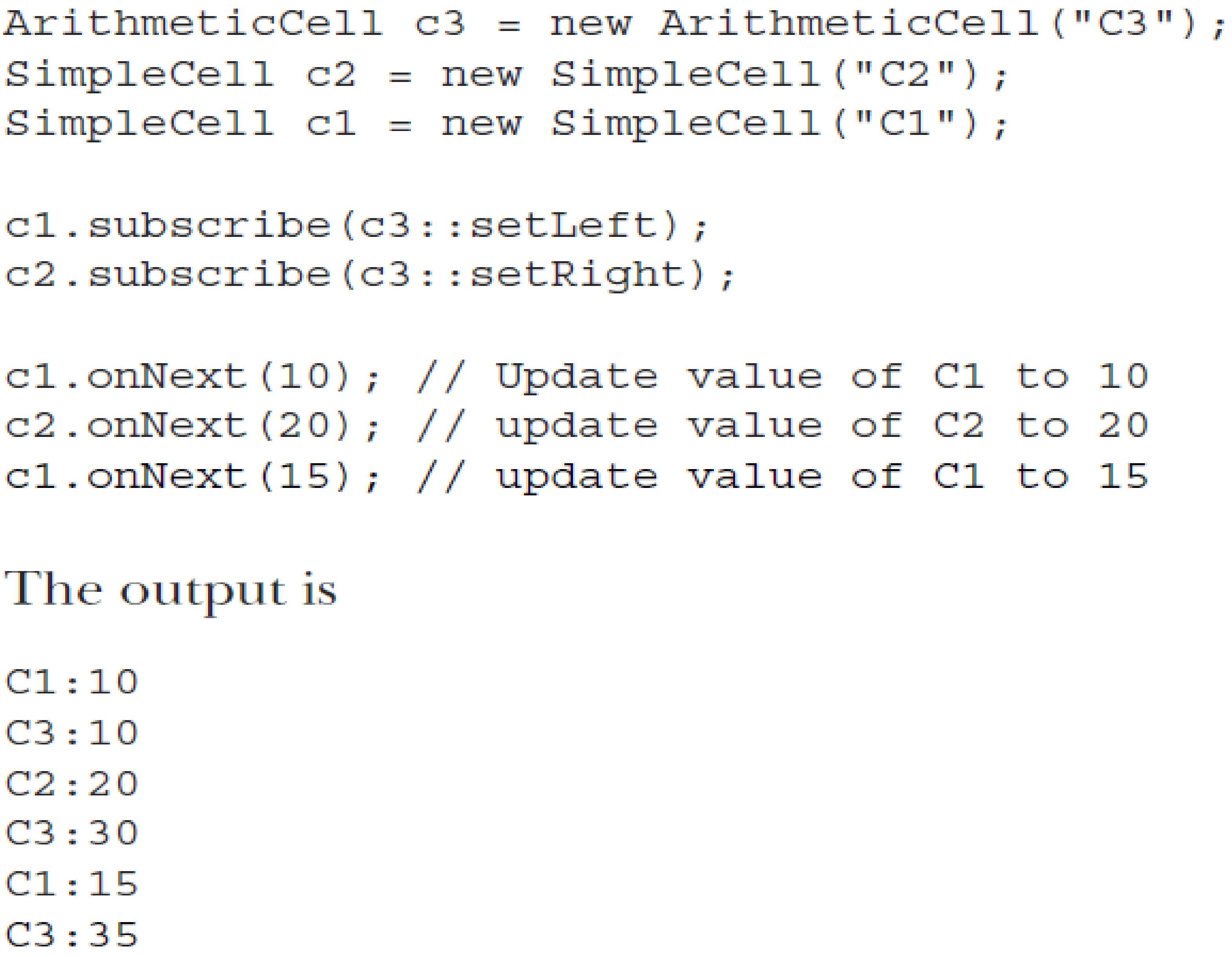


```
public class ArithmeticCell extends SimpleCell {

    private int left;
    private int right;

    public ArithmeticCell(String name) {
        super(name);
    }

    public void setLeft(int left) {
        this.left = left;
        onNext(left + this.right);
    }

    public void setRight(int right) {
        this.right = right;
        onNext(right + this.left);
    }
}
```

Update the cell value and notify any subscribers.

Update the cell value and notify any subscribers.

Reactive Programming:Simple Example-2

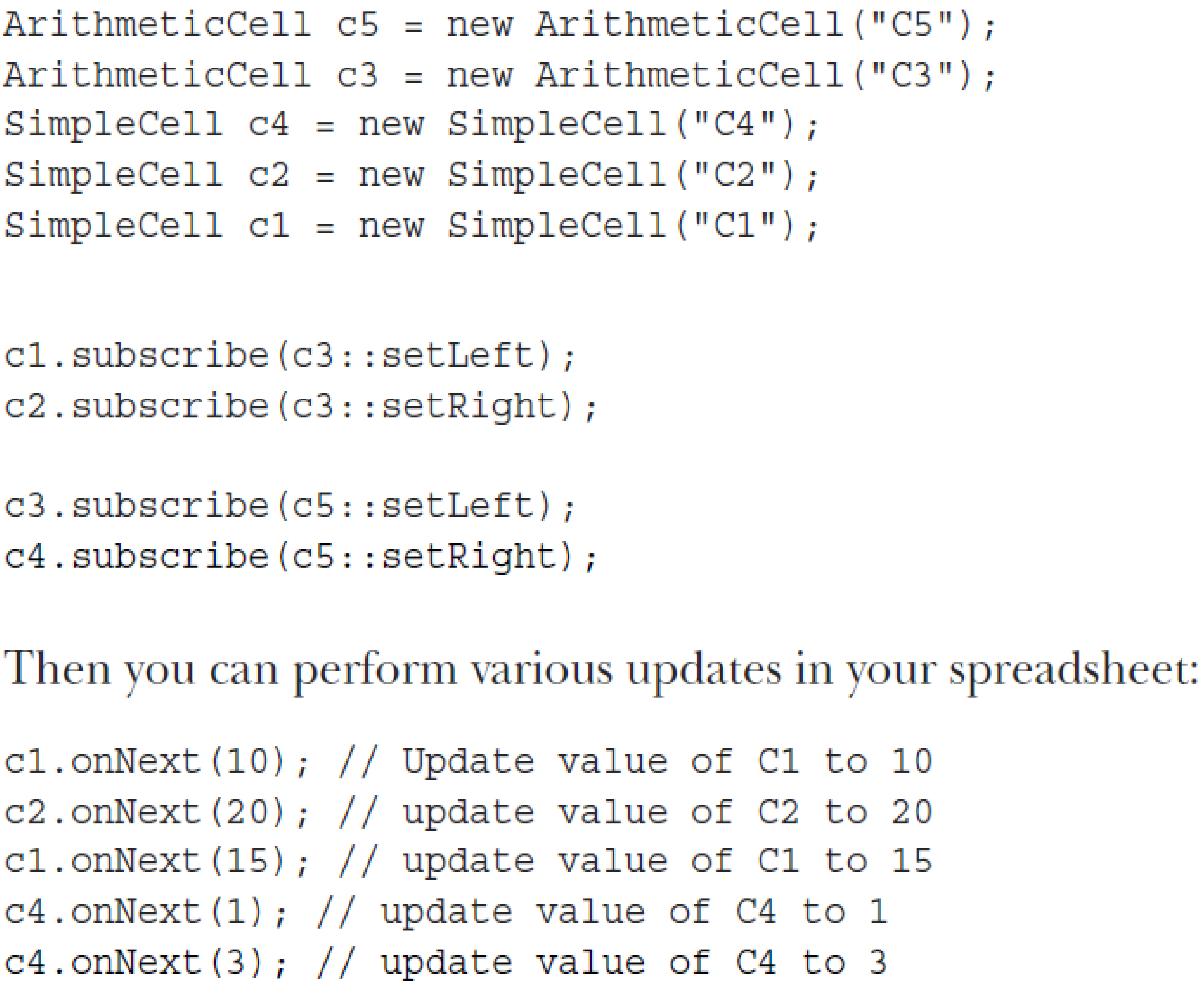


```
ArithmeticCell c3 = new ArithmeticCell("C3");
SimpleCell c2 = new SimpleCell("C2");
SimpleCell c1 = new SimpleCell("C1");

c1.subscribe(c3::setLeft);
c2.subscribe(c3::setRight);

c1.onNext(10); // Update value of C1 to 10
c2.onNext(20); // update value of C2 to 20
c1.onNext(15); // update value of C1 to 15
```

The output is

```
C1:10
C3:10
C2:20
C3:30
C1:15
C3:35
```

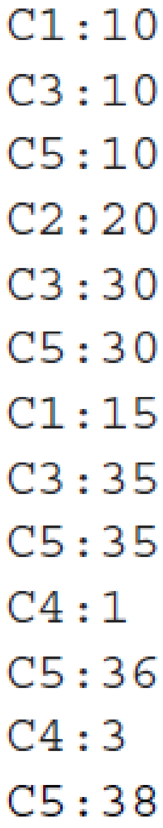
Reactive Programming:Simple Example-2



```
ArithmeticCell c5 = new ArithmeticCell("C5");
ArithmeticCell c3 = new ArithmeticCell("C3");
SimpleCell c4 = new SimpleCell("C4");
SimpleCell c2 = new SimpleCell("C2");
SimpleCell c1 = new SimpleCell("C1");


c1.subscribe(c3::setLeft);
c2.subscribe(c3::setRight);

c3.subscribe(c5::setLeft);
c4.subscribe(c5::setRight);
```

Then you can perform various updates in your spreadsheet:

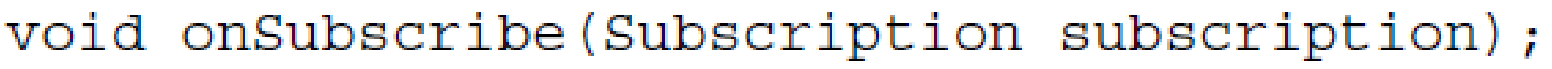
```
c1.onNext(10); // Update value of C1 to 10
c2.onNext(20); // update value of C2 to 20
c1.onNext(15); // update value of C1 to 15
c4.onNext(1); // update value of C4 to 1
c4.onNext(3); // update value of C4 to 3
```



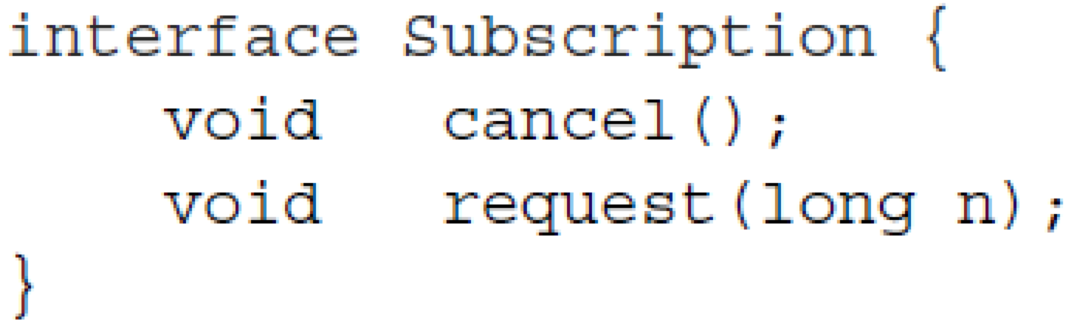
```
C1:10
C3:10
C5:10
C2:20
C3:30
C5:30
C1:15
C3:35
C5:35
C4:1
C5:36
C4:3
C5:38
```

Reactive Programming:Flow:Back pressure

- You want to limit the rate at which this information is sent via backpressure (flow control), which requires you to send information from Subscriber to Publisher.
  - The problem is that the Publisher may have multiple Subscribers, and you want backpressure to affect only the point-to-point connection involved.
  - In the Java 9 Flow API, the Subscriber interface includes a fourth method



```
void onSubscribe(Subscription subscription);
```



```
interface Subscription {
    void   cancel();
    void   request(long n);
}
```

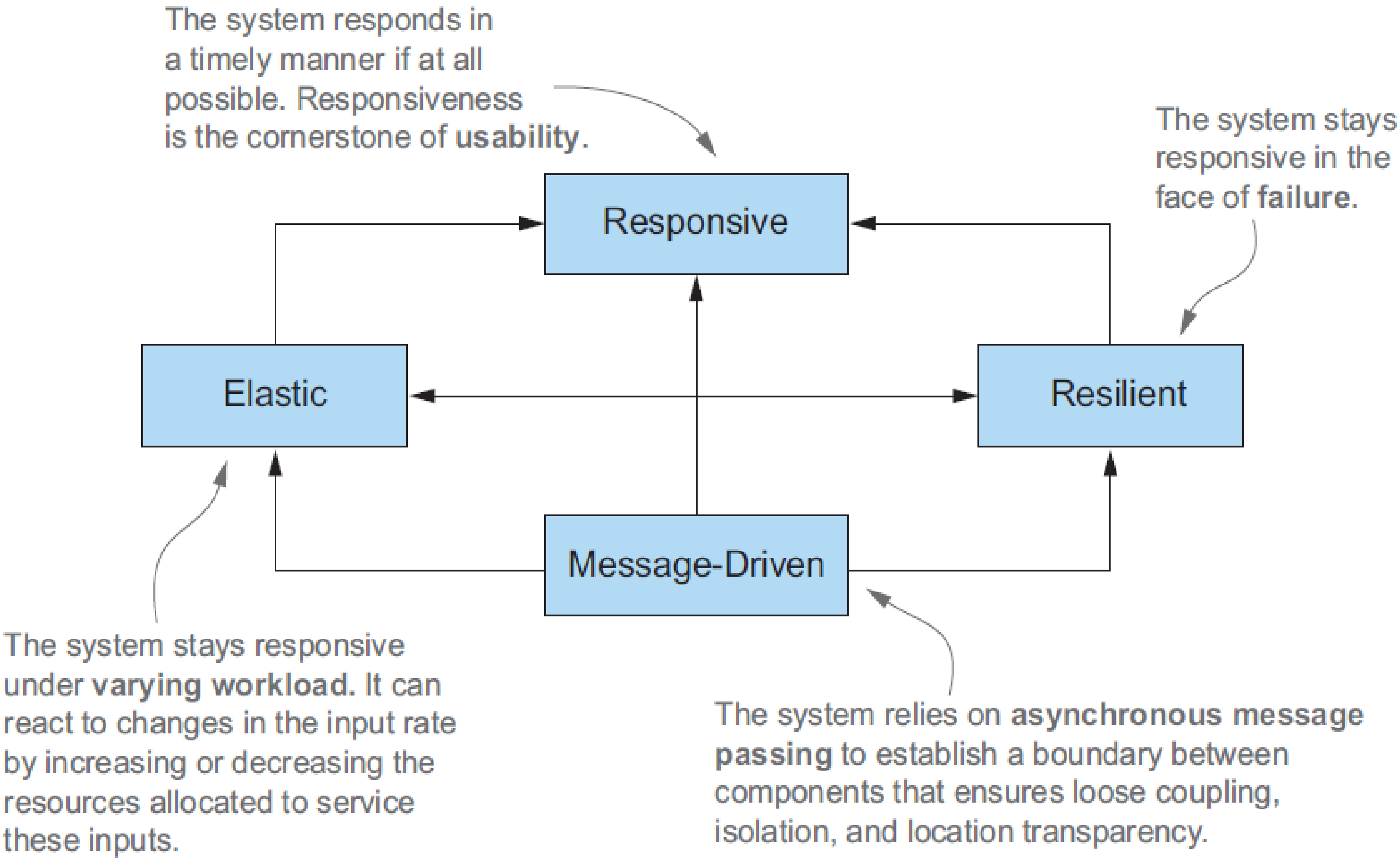
Reactive Programming:Flow:Back pressure

- To enable a publish-subscribe connection to deal with events one at a time, you need to make the following changes:
    - Arrange for the Subscriber to store the Subscription object passed by OnSubscribe locally, perhaps as a field subscription.
    - Make the last action of onSubscribe, onNext, and (perhaps) onError be a call to channel.request(1) to request the next event (only one event, which stops the Subscriber from being overwhelmed).
    - Change the Publisher so that notifyAllSubscribers (in this example) sends an onNext or onError event along only the channels that made a request.
        - (Typically, the Publisher creates a new Subscription object to associate with each Subscriber so that multiple Subscribers can each process data at their own rate.)

Reactive Programming:Flow:Back pressure

- Although this process seems to be simple, implementing backpressure requires thinking about a range of implementation trade-offs:
  - Do you send events to multiple Subscribers at the speed of the slowest, or do you have a separate queue of as-yet-unsent data for each Subscriber?
  - What happens when these queues grow excessively?
  - Do you drop events if the Subscriber isn’t ready for them?

Reactive Programming:Manifesto



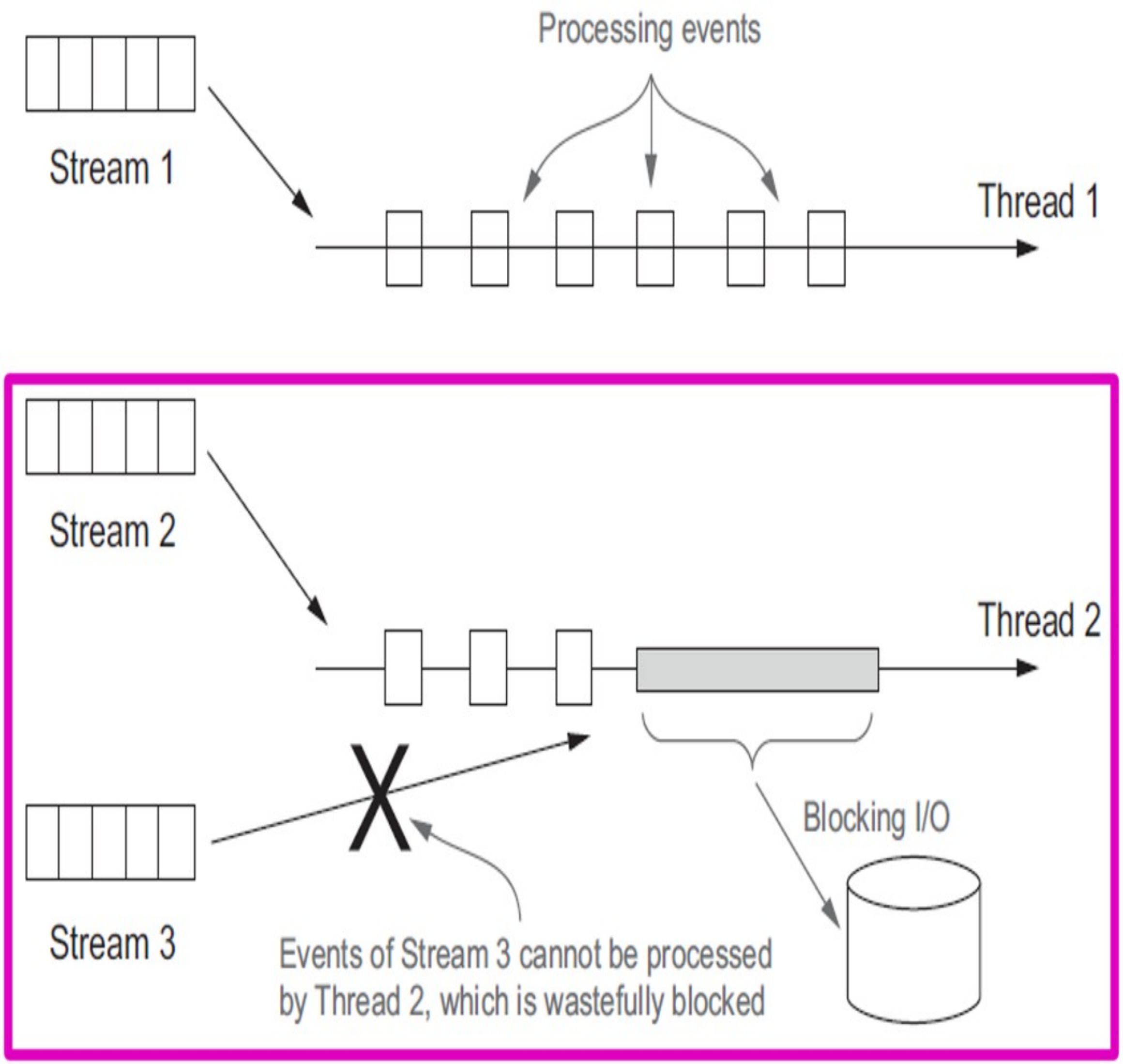
The system responds in a timely manner if at all possible. Responsiveness is the cornerstone of **usability**.

The system stays responsive in the face of **failure**.

The system stays responsive under **varying workload.** It can react to changes in the input rate by increasing or decreasing the resources allocated to service these inputs.

The system relies on **asynchronous message passing** to establish a boundary between components that ensures loose coupling, isolation, and location transparency.

Reactive Programming:Manifesto



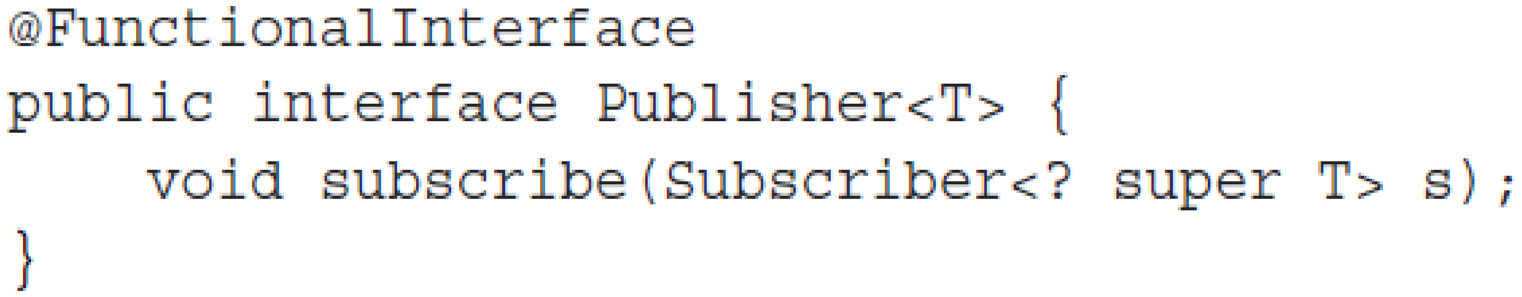
Events of Stream 3 cannot be processed by Thread 2, which is wastefully blocked

- Thread/stream processing may trigger blocking IO which hinders processing an available stream despite the thread being free (blocked)
- To overcome this problem, frameworks like RxJava and Akka, allow blocking operations to excuted on seperate dedicated thread pool.

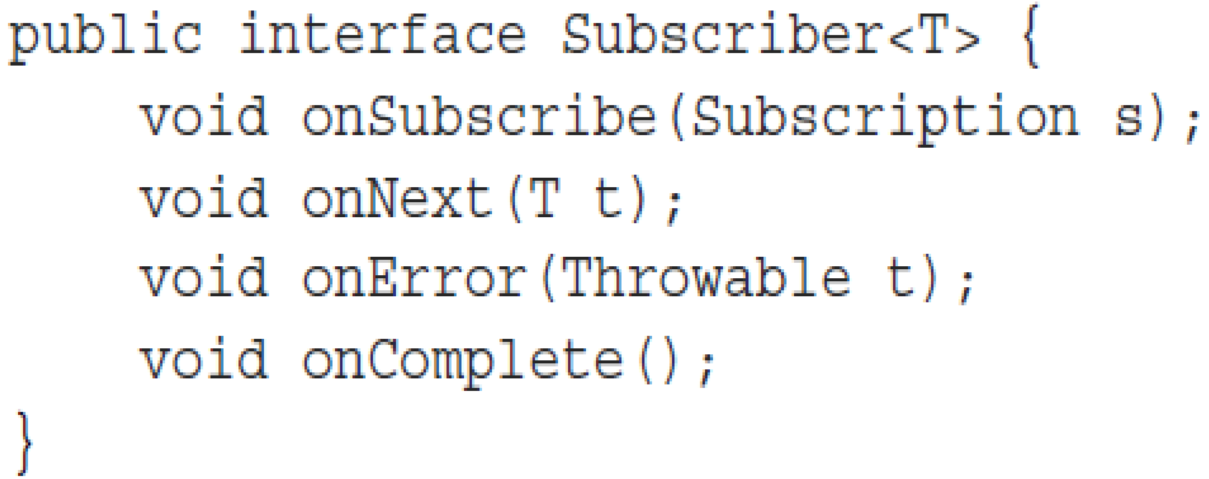
Reactive Programming:Flow

- Java 9 adds one new class for reactive programming: java.util.concurrent.Flow.
  - This class contains only static components and can’t be instantiated. The Flow class contains four nested interfaces to express the publish-subscribe model of reactive programming as standardized by the Reactive Streams project:
    - Publisher
    - Subscriber
    - Subscription
    - Processor

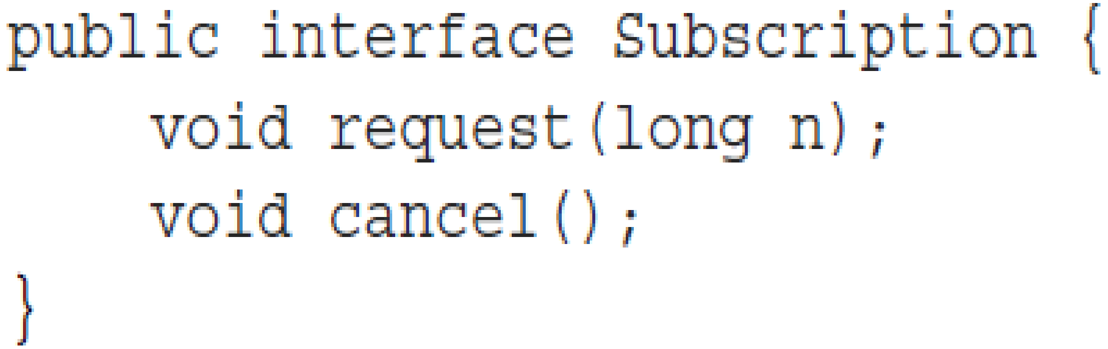
Reactive Programming:Flow



```
@FunctionalInterface
public interface Publisher<T> {
    void subscribe(Subscriber<? super T> s);
}
```



```
public interface Subscriber<T> {
    void onSubscribe(Subscription s);
    void onNext(T t);
    void onError(Throwable t);
    void onComplete();
}
```



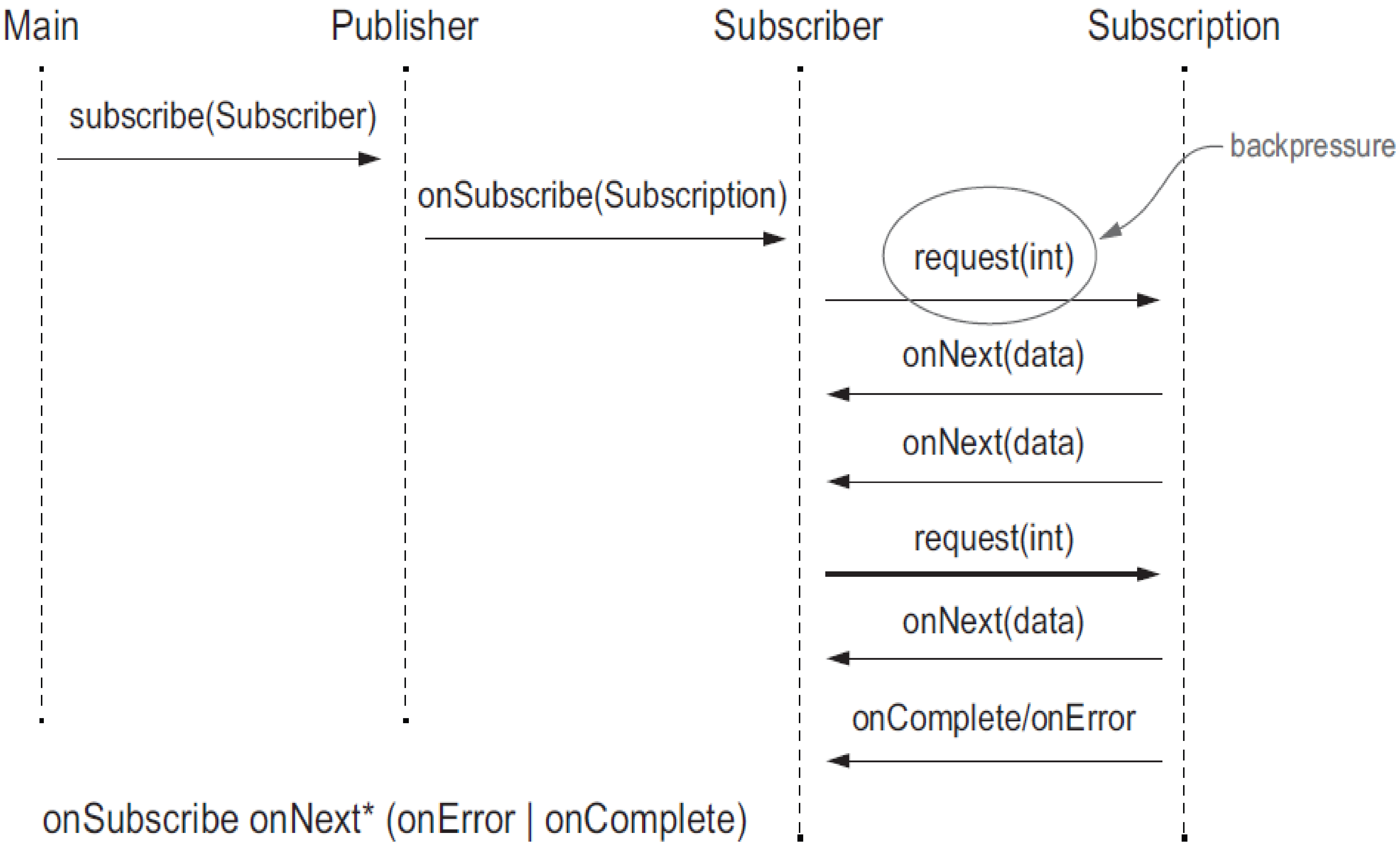
```
public interface Subscription {
    void request(long n);
    void cancel();
}
```

Reactive Programming:Flow

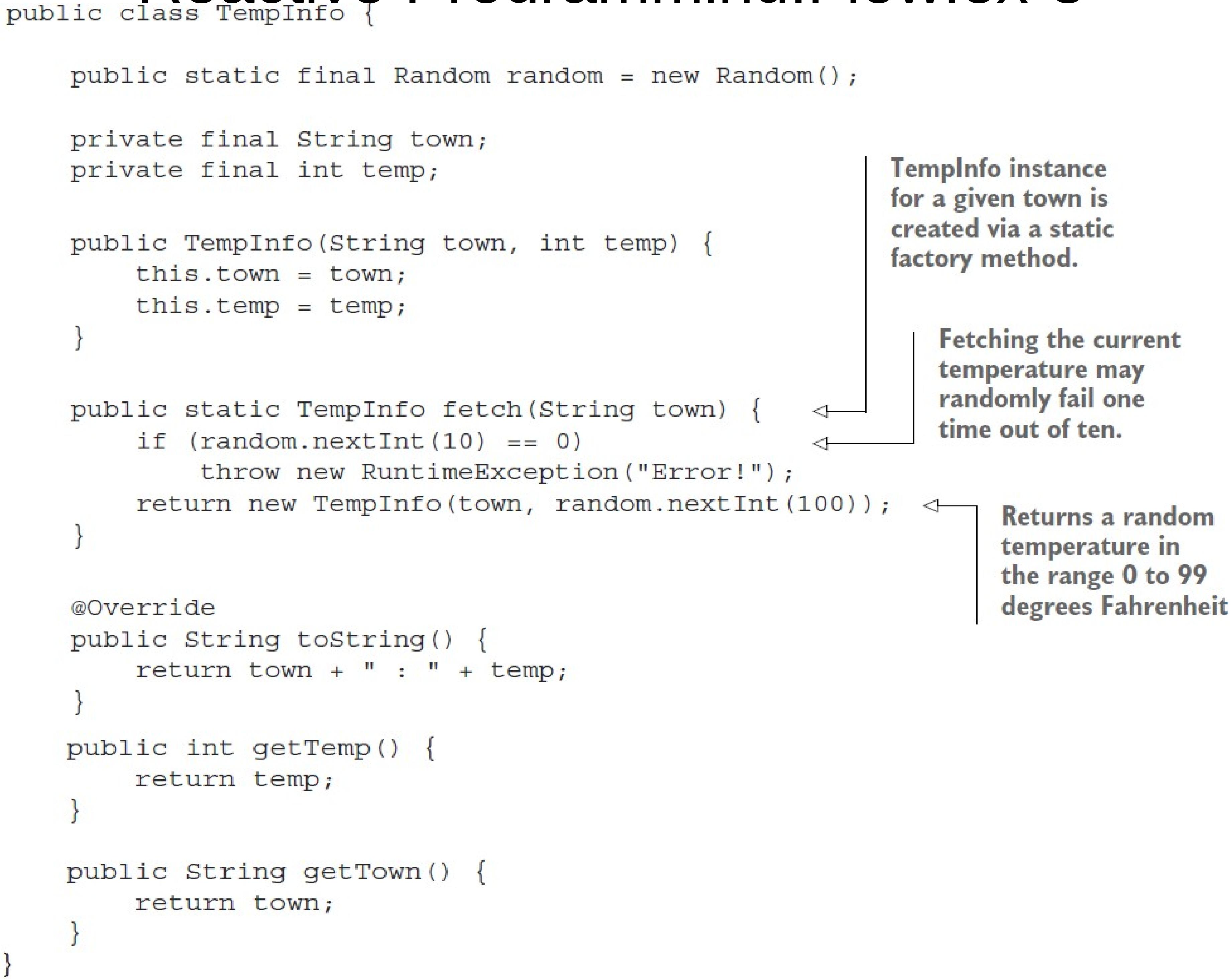
- Flow for Flow
  - This notation means that onSubscribe is always invoked as the first event, followed by an arbitrary number of onNext signals.
  - The stream of events can go on forever, or it can be terminated by an onComplete callback to signify that no more elements will be produced or by an onError if the Publisher experiences a failure.

Reactive Programming:Flow

- Flow for Flow



Reactive Programming:Flow:ex-3



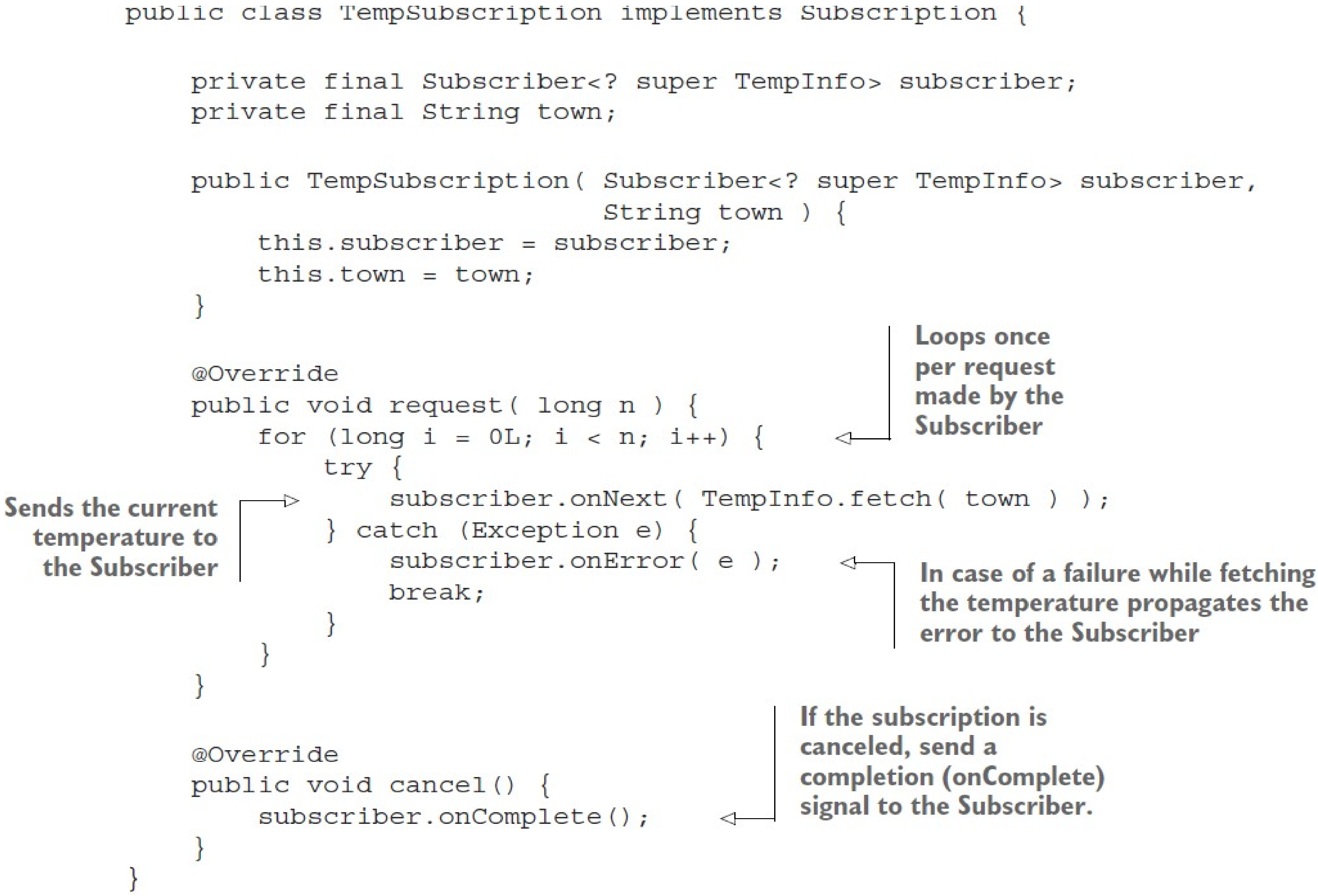
```
public class TempInfo {

    public static final Random random = new Random();

    private final String town;
    private final int temp;

    public TempInfo(String town, int temp) {
        this.town = town;
        this.temp = temp;
    }

    public static TempInfo fetch(String town) {
        if (random.nextInt(10) == 0)
            throw new RuntimeException("Error!");
        return new TempInfo(town, random.nextInt(100));
    }

    @Override
    public String toString() {
        return town + " : " + temp;
    }
    public int getTemp() {
        return temp;
    }

    public String getTown() {
        return town;
    }
}
```

TempInfo instance for a given town is created via a static factory method.

Fetching the current temperature may randomly fail one time out of ten.

Returns a random temperature in the range 0 to 99 degrees Fahrenheit

Reactive Programming:Flow:ex-3



```java
public class TempSubscription implements Subscription {

    private final Subscriber<? super TempInfo> subscriber;
    private final String town;

    public TempSubscription( Subscriber<? super TempInfo> subscriber,
                             String town ) {
        this.subscriber = subscriber;
        this.town = town;
    }

    @Override
    public void request( long n ) {
        for (long i = 0L; i < n; i++) {
            try {
                subscriber.onNext( TempInfo.fetch( town ) );
            } catch (Exception e) {
                subscriber.onError( e );
                break;
            }
        }
    }

    @Override
    public void cancel() {
        subscriber.onComplete();
    }
}
```

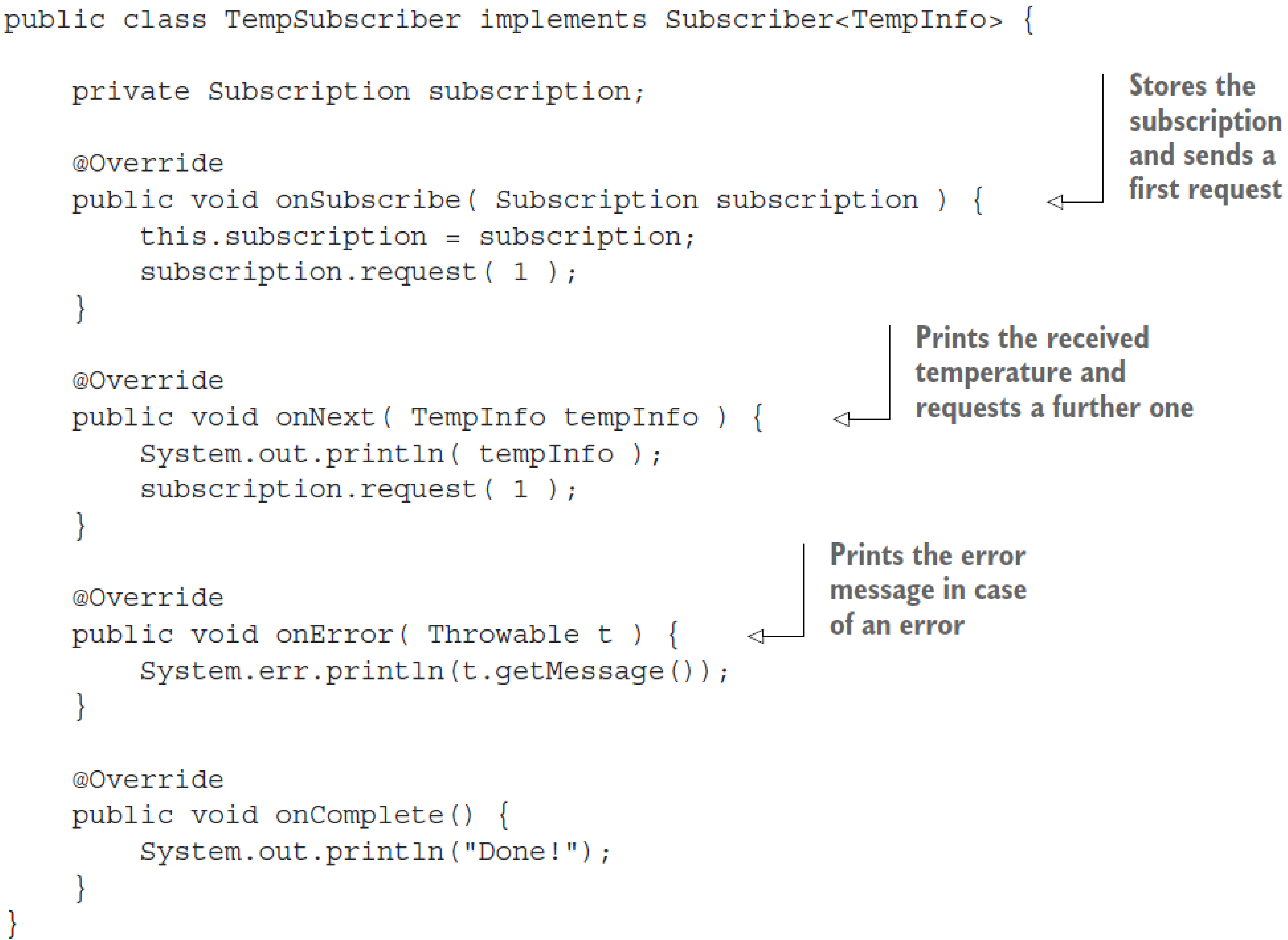
Loops once per request made by the Subscriber

Sends the current temperature to the Subscriber

In case of a failure while fetching the temperature propagates the error to the Subscriber

If the subscription is canceled, send a completion (onComplete) signal to the Subscriber.

Reactive Programming:Flow:ex-3



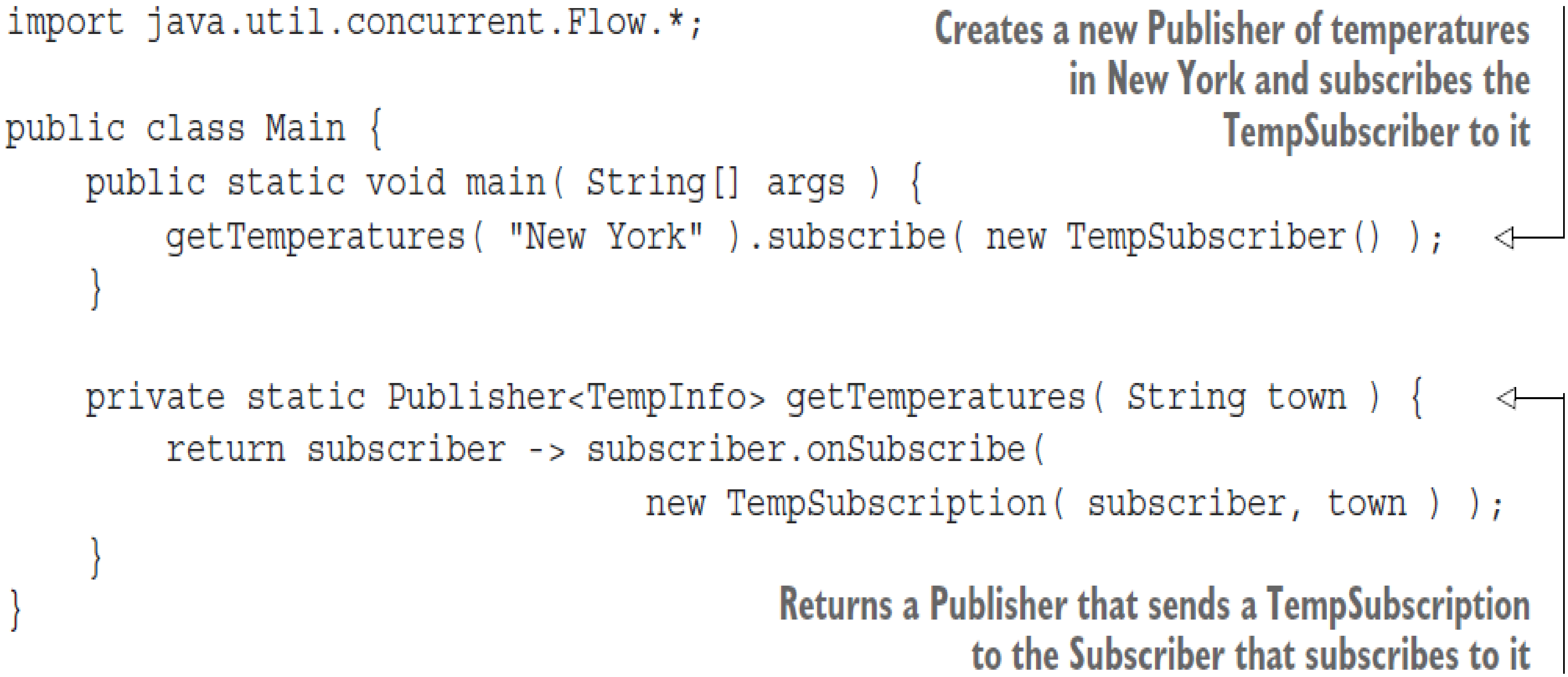
```
public class TempSubscriber implements Subscriber<TempInfo> {

    private Subscription subscription;

    @Override
    public void onSubscribe( Subscription subscription ) {
        this.subscription = subscription;
        subscription.request( 1 );
    }

    @Override
    public void onNext( TempInfo tempInfo ) {
        System.out.println( tempInfo );
        subscription.request( 1 );
    }

    @Override
    public void onError( Throwable t ) {
        System.err.println(t.getMessage());
    }

    @Override
    public void onComplete() {
        System.out.println("Done!");
    }
}
```

Stores the subscription and sends a first request

Prints the received temperature and requests a further one

Prints the error message in case of an error

Reactive Programming:Flow:ex-3

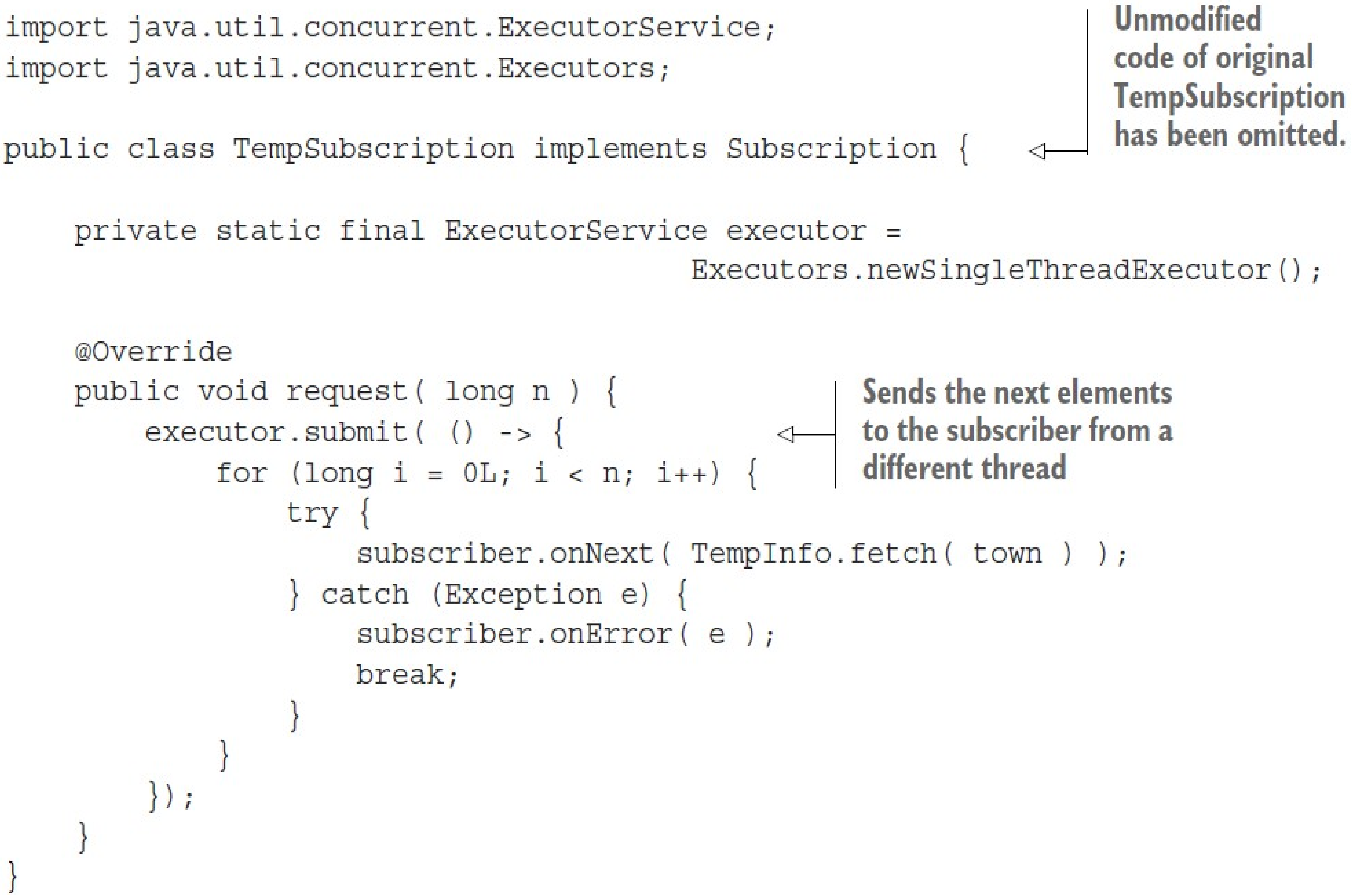


```
import java.util.concurrent.Flow.*;

public class Main {
    public static void main( String[] args ) {
        getTemperatures( "New York" ).subscribe( new TempSubscriber() );
    }

    private static Publisher<TempInfo> getTemperatures( String town ) {
        return subscriber -> subscriber.onSubscribe(
                                new TempSubscription( subscriber, town ) );
    }
}
```

Creates a new Publisher of temperatures in New York and subscribes the TempSubscriber to it

Returns a Publisher that sends a TempSubscription to the Subscriber that subscribes to it

Reactive Programming:Flow:ex-3

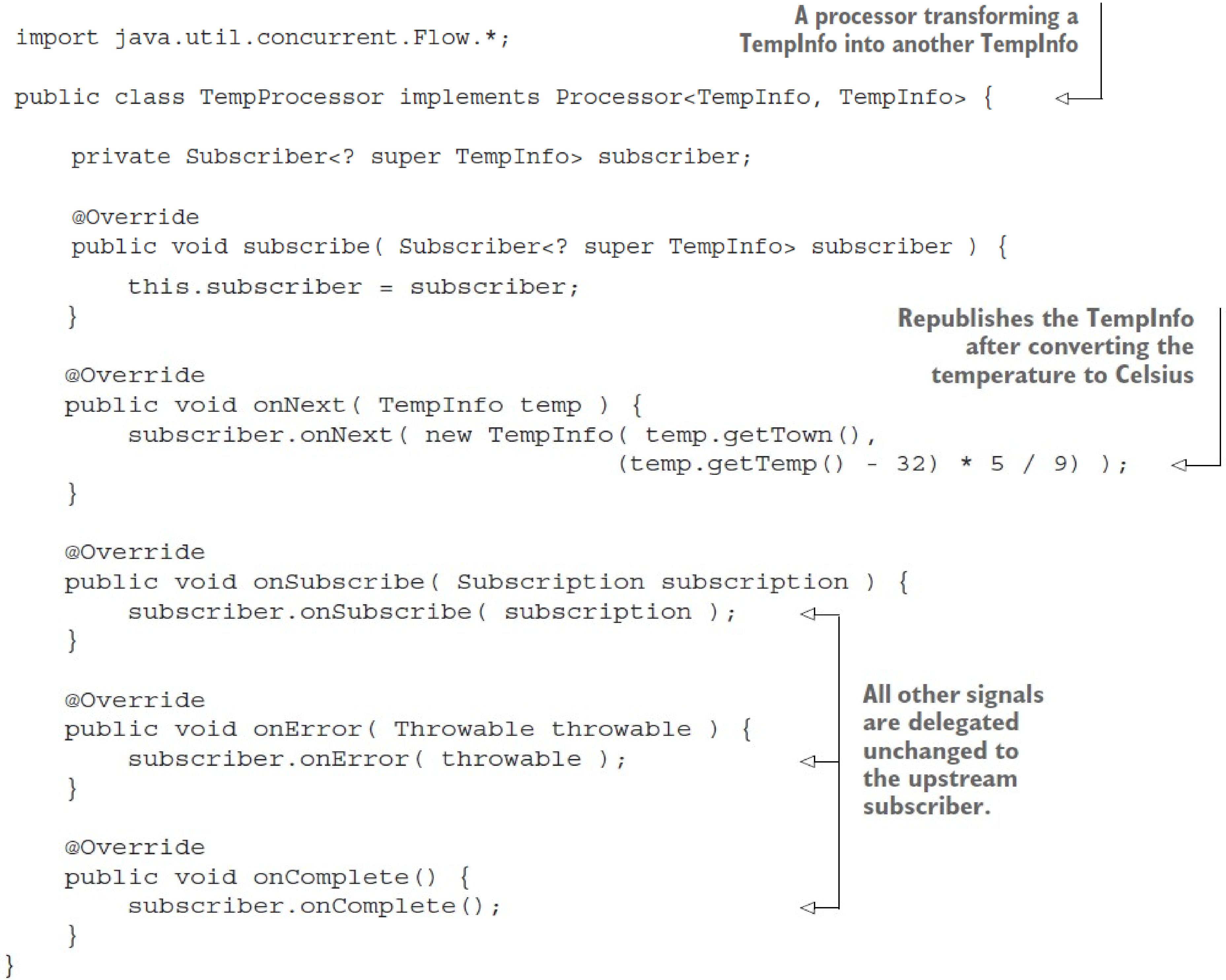


```
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;

public class TempSubscription implements Subscription {

    private static final ExecutorService executor =
                                       Executors.newSingleThreadExecutor();

    @Override
    public void request( long n ) {
        executor.submit( () -> {
            for (long i = 0L; i < n; i++) {
                try {
                    subscriber.onNext( TempInfo.fetch( town ) );
                } catch (Exception e) {
                    subscriber.onError( e );
                    break;
                }
            }
        });
    }
}
```

Unmodified code of original TempSubscription has been omitted.

Sends the next elements to the subscriber from a different thread

Reactive Programming:Flow:ex-4



```
import java.util.concurrent.Flow.*;

public class TempProcessor implements Processor<TempInfo, TempInfo> {

    private Subscriber<? super TempInfo> subscriber;

    @Override
    public void subscribe( Subscriber<? super TempInfo> subscriber ) {
        this.subscriber = subscriber;
    }

    @Override
    public void onNext( TempInfo temp ) {
        subscriber.onNext( new TempInfo( temp.getTown(),
                                         (temp.getTemp() - 32) * 5 / 9) );
    }

    @Override
    public void onSubscribe( Subscription subscription ) {
        subscriber.onSubscribe( subscription );
    }

    @Override
    public void onError( Throwable throwable ) {
        subscriber.onError( throwable );
    }

    @Override
    public void onComplete() {
        subscriber.onComplete();
    }
}
```

A processor transforming a TempInfo into another TempInfo

Republishes the TempInfo after converting the temperature to Celsius

All other signals are delegated unchanged to the upstream subscriber.

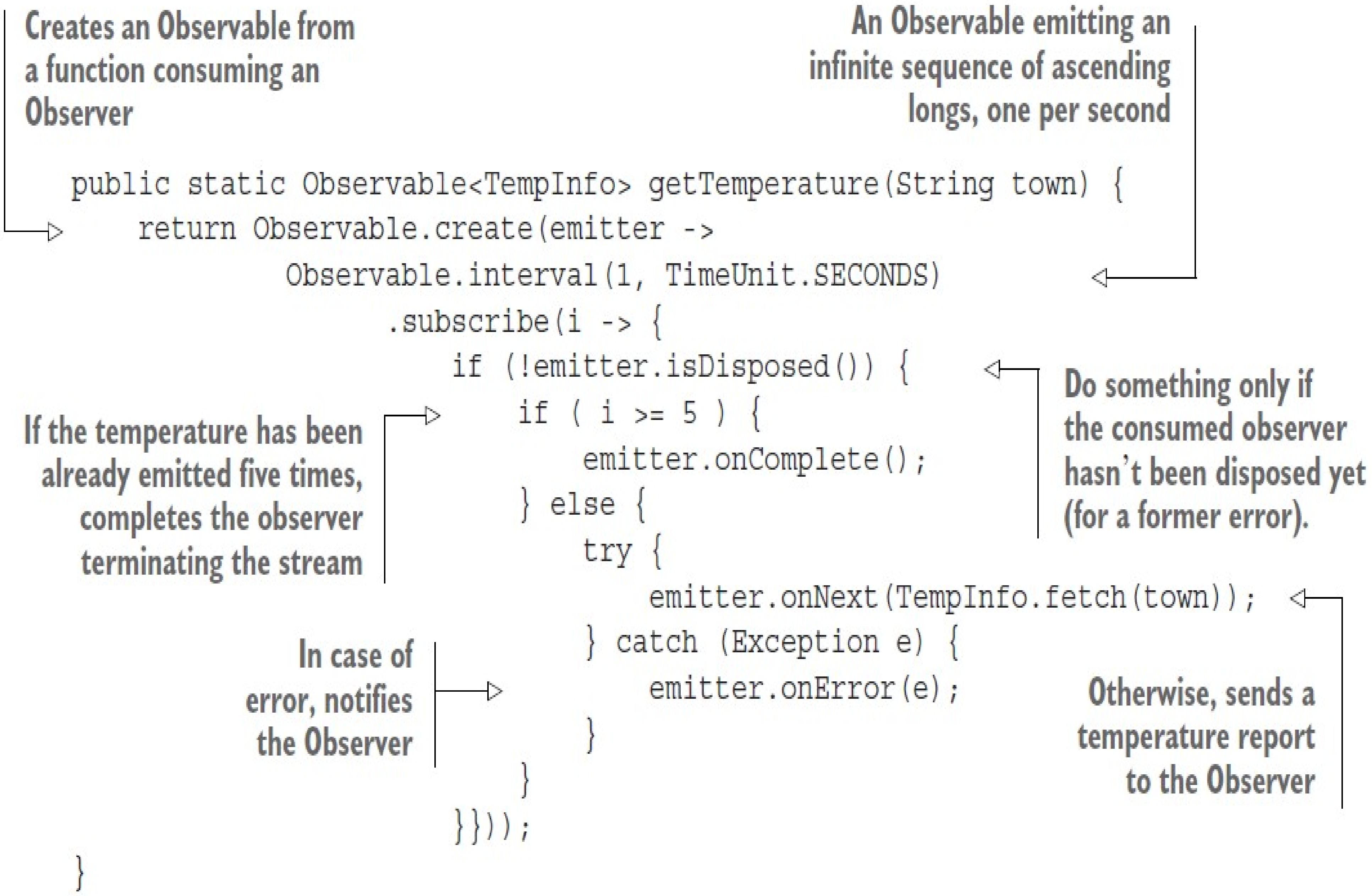
Reactive Programming:RxJava

- On reading the RxJava documentation, you find that one class is the io.reactivex.Flowable class, which includes the **reactive pull-based backpressure feature of Java 9 Flow**.
  - Backpressure prevents a Subscriber from being overrun by data being produced by a fast Publisher.
- The other class is the original **RxJava** io.reactivex.Observable **version of Publisher, which didn’t support backpressure.**
  - This class is both simpler to program and more appropriate for user-interface events (such as mouse movements);
  - these events are streams that can’t be reasonably backpressured. (You can’t ask the user to slow down or stop moving the mouse!)
  - For this reason, RxJava provides these two implementing classes for the common idea stream of events.

Reactive Programming:RxJava

- The RxJava advice is to use the nonbackpressured Observable when you have a stream of no more than a thousand elements or when you’re are dealing with GUI events such as mouse moves or touch events, which are impossible to backpressure and aren’t frequent anyway.
- It’s worth noting that any subscriber can **effectively turn off backpressuring by invoking request(Long.MAX_VALUE)**on the subscription, even if this practice isn’t advisable unless you’re sure that the Subscriber will always be able to process all the received events in a timely manner.

Reactive Programming:RxJava:ex-1



Creates an Observable from a function consuming an Observer

An Observable emitting an infinite sequence of ascending longs, one per second

```
public static Observable<TempInfo> getTemperature(String town) {
    return Observable.create(emitter ->
             Observable.interval(1, TimeUnit.SECONDS)
                   .subscribe(i -> {
                       if (!emitter.isDisposed()) {
                           if ( i >= 5 ) {
                               emitter.onComplete();
                           } else {
                               try {
                                   emitter.onNext(TempInfo.fetch(town));
                               } catch (Exception e) {
                                   emitter.onError(e);
                               }
                           }
                       }}));
}
```

Do something only if the consumed observer hasn't been disposed yet (for a former error).

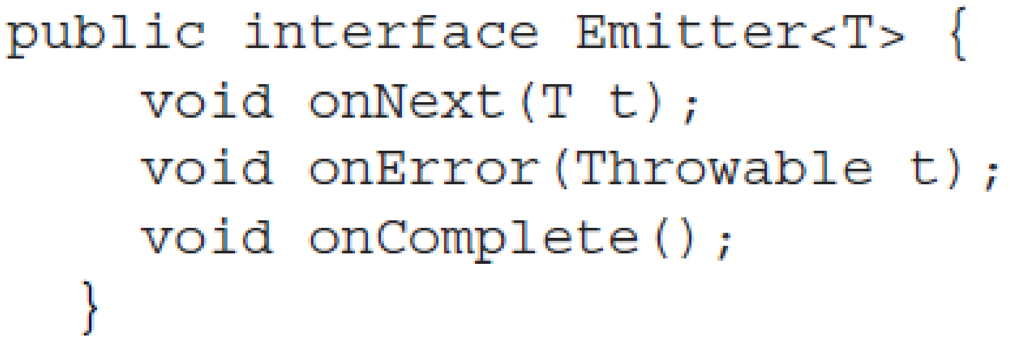
If the temperature has been already emitted five times, completes the observer terminating the stream

Otherwise, sends a temperature report to the Observer

In case of error, notifies the Observer

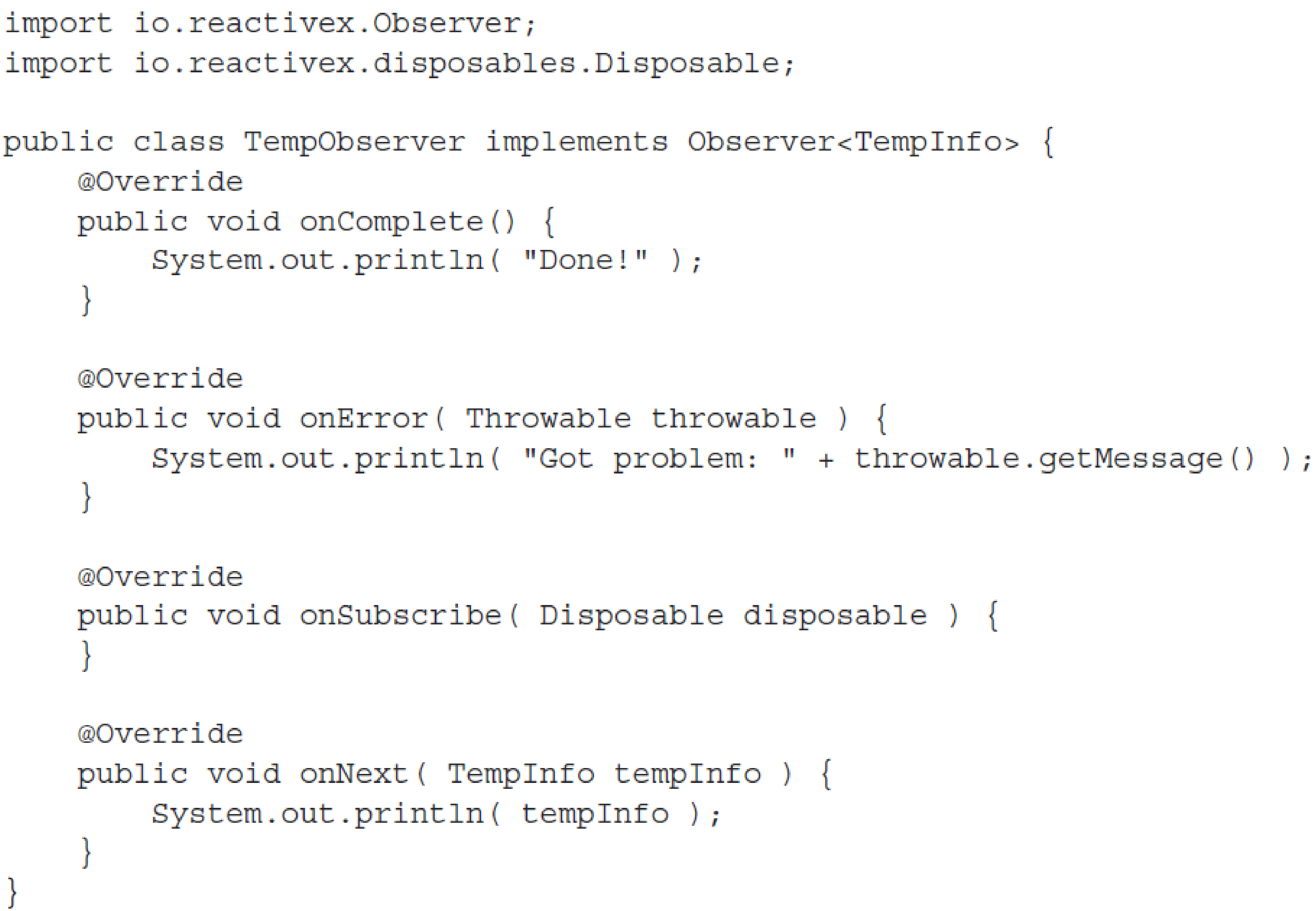
Reactive Programming:RxJava:ex-1

- The RxJava ObservableEmitter interface extends the basic RxJava Emitter, which you can think of as being an Observer without the onSubscribe method.



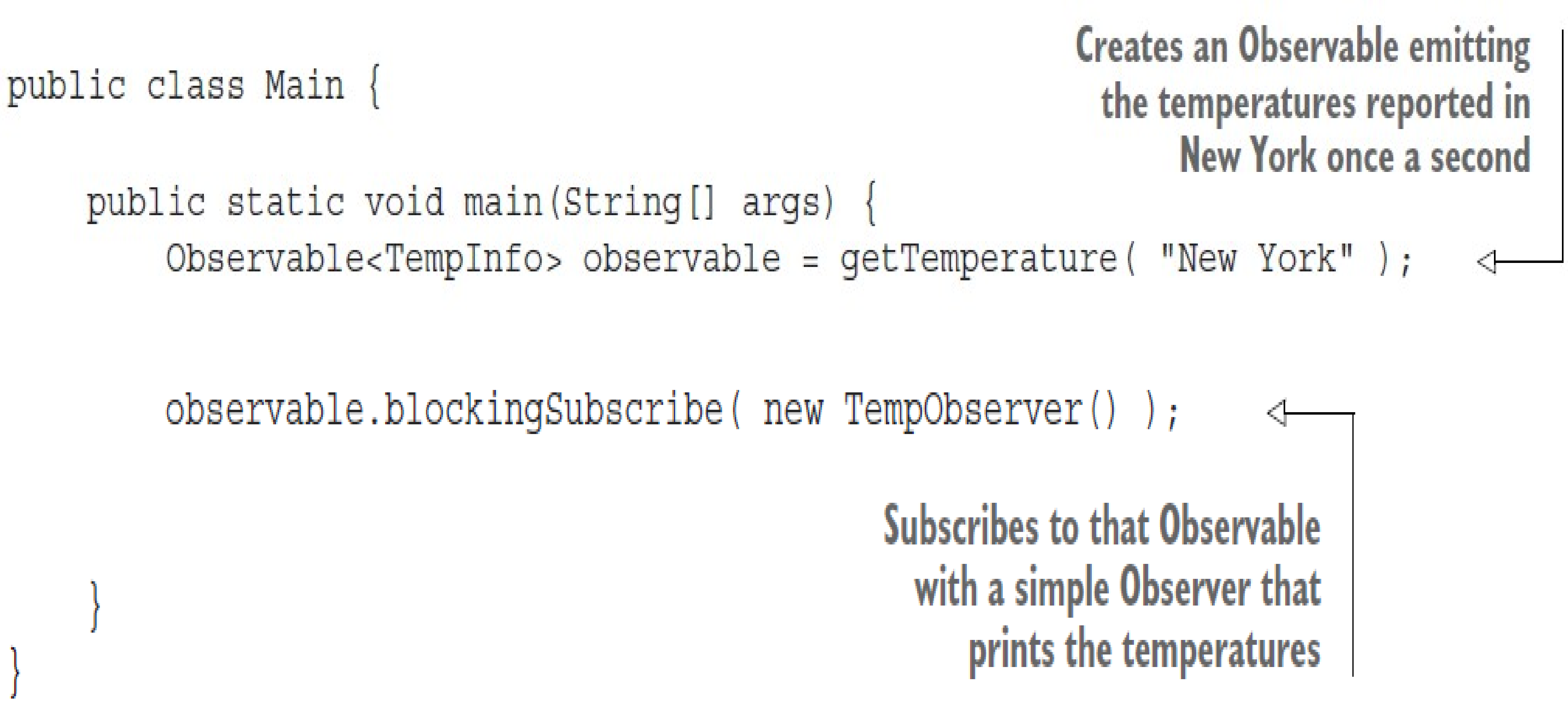
```
public interface Emitter<T> {
    void onNext(T t);
    void onError(Throwable t);
    void onComplete();
  }
```

Reactive Programming:RxJava:ex-1



```
import io.reactivex.Observer;
import io.reactivex.disposables.Disposable;

public class TempObserver implements Observer<TempInfo> {
    @Override
    public void onComplete() {
        System.out.println( "Done!" );
    }

    @Override
    public void onError( Throwable throwable ) {
        System.out.println( "Got problem: " + throwable.getMessage() );
    }

    @Override
    public void onSubscribe( Disposable disposable ) {
    }

    @Override
    public void onNext( TempInfo tempInfo ) {
        System.out.println( tempInfo );
    }
}
```

Reactive Programming:RxJava:ex-1



```
public class Main {

    public static void main(String[] args) {
        Observable<TempInfo> observable = getTemperature( "New York" );


        observable.blockingSubscribe( new TempObserver() );



    }
}
```

Creates an Observable emitting the temperatures reported in New York once a second

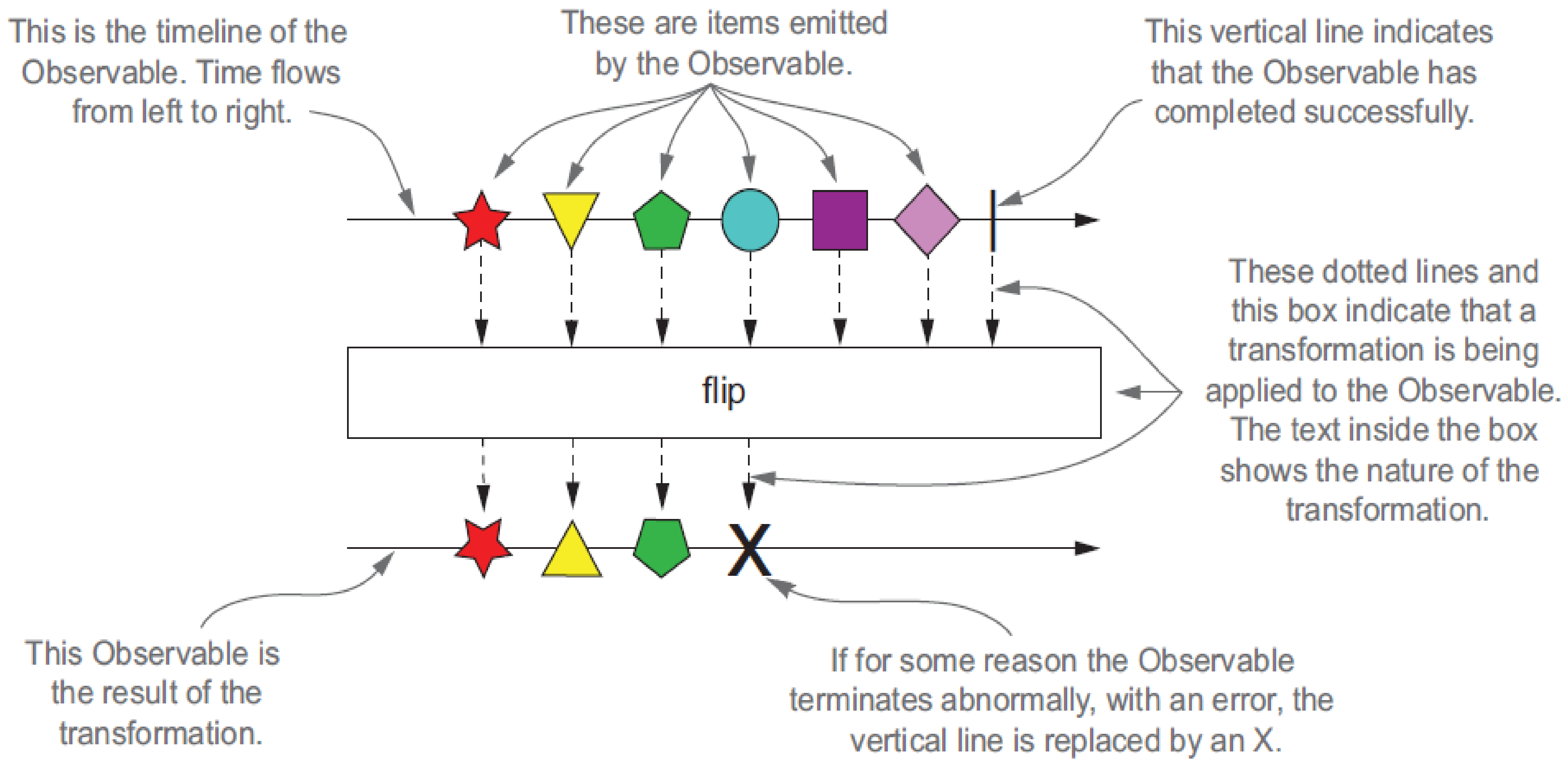
Subscribes to that Observable with a simple Observer that prints the temperatures

Reactive Programming:RxJava

- One of the main advantages of RxJava and other reactive libraries in working with reactive streams, compared with what’s offered by the native Java 9 Flow API, is that they provide a rich toolbox of functions to combine, create, and filter any of those streams.
  - you can also filter a stream to get another one that has only the elements you’re interested in, transform those elements with a given mapping function (both these things can be achieved with Flow.Processor),
  - or even merge or combine two streams in many ways (which can’t be achieved with Flow.Processor).

Reactive Programming:RxJava

- To alleviate the problem of complex functionality like merging streams, the reactive-streams community decided to document the behaviors of these functions in a visual way, using so-called marble diagrams.



This is the timeline of the Observable. Time flows from left to right.

These are items emitted by the Observable.

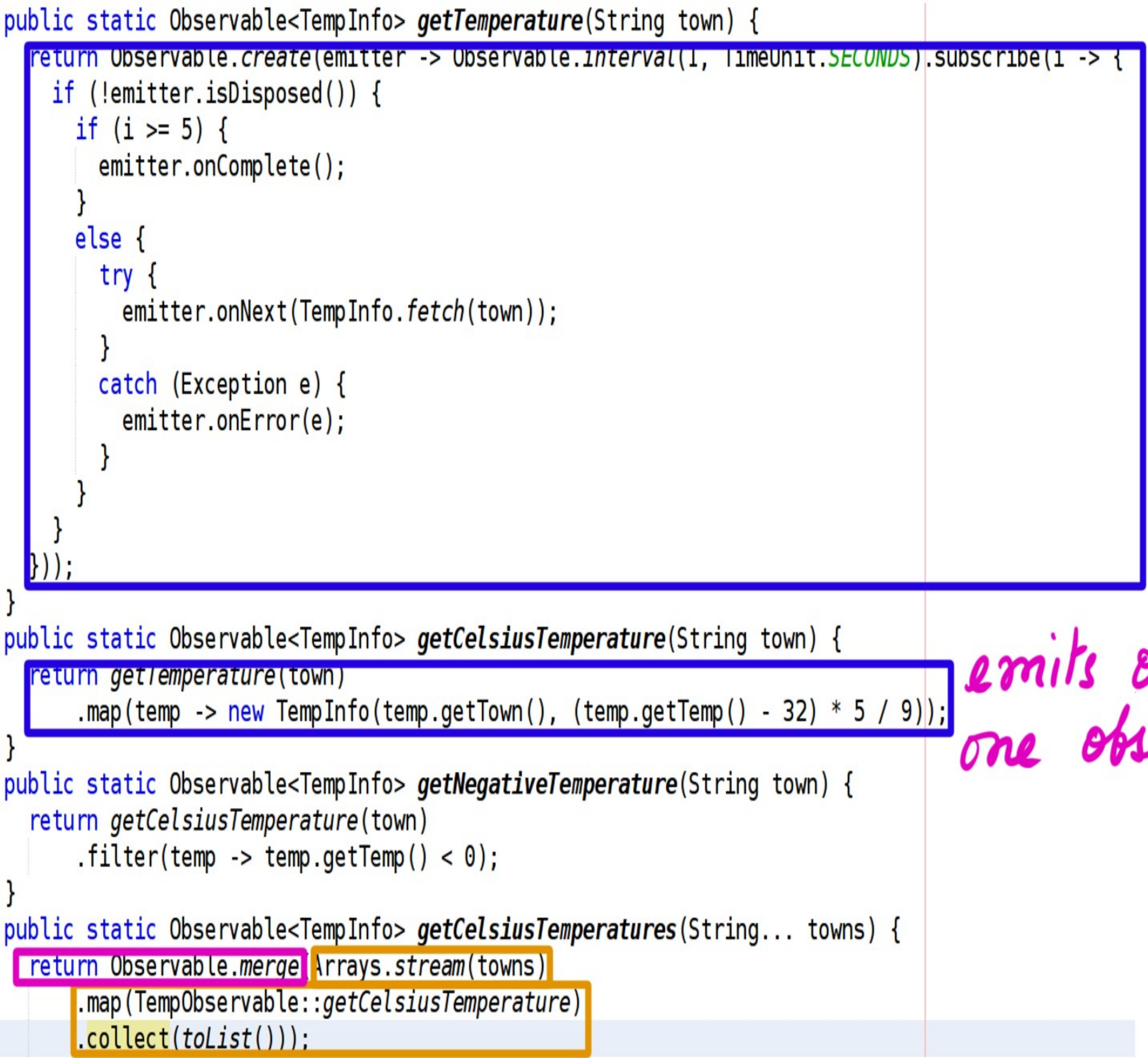
This vertical line indicates that the Observable has completed successfully.

These dotted lines and this box indicate that a transformation is being applied to the Observable. The text inside the box shows the nature of the transformation.

This Observable is the result of the transformation.

If for some reason the Observable terminates abnormally, with an error, the vertical line is replaced by an X.

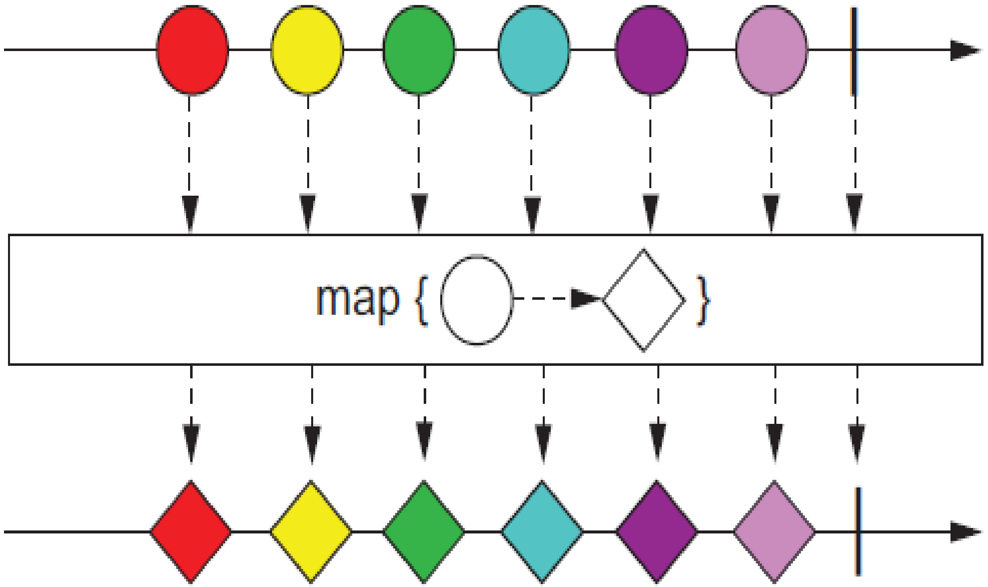
Reactive Programming:RxJava:ex-2

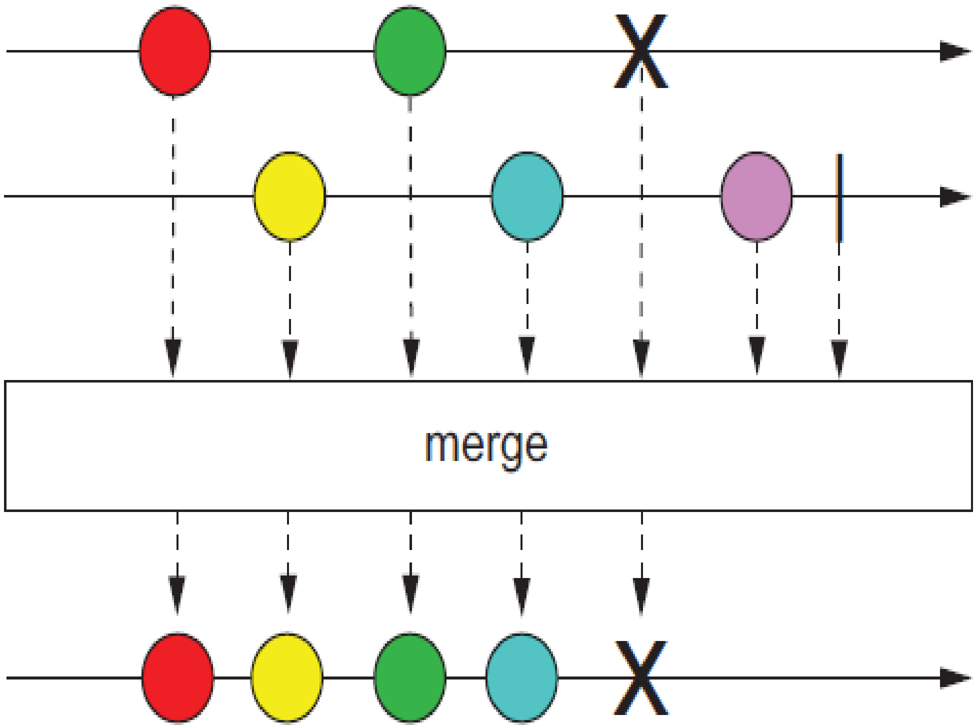


```
public static Observable<TempInfo> getTemperature(String town) {
  return Observable.create(emitter -> Observable.interval(1, TimeUnit.SECONDS).subscribe(i -> {
    if (!emitter.isDisposed()) {
      if (i >= 5) {
        emitter.onComplete();
      }
      else {
        try {
          emitter.onNext(TempInfo.fetch(town));
        }
        catch (Exception e) {
          emitter.onError(e);
        }
      }
    }
  }));
}
public static Observable<TempInfo> getCelsiusTemperature(String town) {
  return getTemperature(town)
      .map(temp -> new TempInfo(temp.getTown(), (temp.getTemp() - 32) * 5 / 9));
}
public static Observable<TempInfo> getNegativeTemperature(String town) {
  return getCelsiusTemperature(town)
      .filter(temp -> temp.getTemp() < 0);
}
public static Observable<TempInfo> getCelsiusTemperatures(String... towns) {
  return Observable.merge(Arrays.stream(towns)
      .map(TempObservable::getCelsiusTemperature)
      .collect(toList()));
```

- Returns an observable of TempInfo.
- Regular Java-8 streams has a list of Observables in the end.
- Flattens an observable that emits observables into one observable.

Reactive Programming:RxJava:ex-2





Flattens an Observable that emits Observables into one Observable, in a way that allows an Observer to receive all successfully emitted items from all of the source Observables without being interrupted by an error notification from one of them, while limiting the number of concurrent subscriptions to these Observables.